In color

Software Design Patterns for Java Developers

2nd Edition

Expert-led approaches to build re-usable software
and enterprise applications

Lalit Mehra

Software Design Patterns for Java Developers

Expert-led approaches to build re-usable software and enterprise applications

2nd Edition

Lalit Mehra

www.bpbonline.com

Second Revised and Updated Edition 2026

First Edition 2022

Copyright © BPB Publications, India

ISBN: 978-93-65899-658

All Rights Reserved. No part of this publication may be reproduced, distributed or transmitted in any form or by any means or stored in a database or retrieval system, without the prior written permission of the publisher with the exception to the program listings which may be entered, stored and executed in a computer system, but they can not be reproduced by the means of publication, photocopy, recording, or by any electronic and mechanical means.

LIMITS OF LIABILITY AND DISCLAIMER OF WARRANTY

The information contained in this book is true and correct to the best of author’s and publisher’s knowledge. The author has made every effort to ensure the accuracy of these publications, but the publisher cannot be held responsible for any loss or damage arising from any information in this book.

All trademarks referred to in the book are acknowledged as properties of their respective owners but BPB Publications cannot guarantee the accuracy of this information.

To View Complete
BPB Publications Catalogue
Scan the QR Code:

www.bpbonline.com

Dedicated to

My parents and my wife, ***Harshita***

About the Author

Lalit Mehra is a software engineer with over fifteen years of experience designing, scaling, and evolving complex software systems. Much of his career has been centered around payments and the broader fintech ecosystem. He currently works as a technical architect at Fyscal Technologies. Previously, he held key engineering roles at Salesforce and Paytm, contributing to large-scale systems operating at significant scale. Across these roles, he has helped design distributed systems that must perform consistently under real-world load and business pressure. His technical focus includes distributed systems, microservices, scalability, and asynchronous architectures. Lalit holds a master's degree in computer science from Indraprastha University, Delhi, and has earned professional certifications from Oracle and Microsoft. Through his writing, he aims to bring clarity to software architecture by grounding it in practical trade-offs rather than abstract theory.

Acknowledgement

I want to express my sincere gratitude to my family and friends for their unrelenting support and encouragement throughout the writing of this book, particularly my wife, Harshita.

BPB Publications were instrumental in helping me bring this book to fruition, and I am grateful for their guidance and expertise. It took a long time to revise this book, with the valuable participation and collaboration of reviewers, technical experts, and editors.

I want to thank my colleagues and co-workers for their valuable contributions during my many years working in the tech industry, who have taught me a lot and provided valuable feedback on my work.

Finally, I would like to express my gratitude to all the readers who have been interested in my book and for their assistance in making it a reality. I am grateful for your encouragement.

Preface

Software design patterns have been around for decades, yet they remain just as relevant today as they were when they were first documented. If you have spent any time building real systems, you have already encountered them. Sometimes you recognized them by name. Often, you simply solve a problem and arrive at a familiar structure without realizing it has a label.

This book explores the three major categories of object-oriented design patterns, which include creational, structural, and behavioral. It is organized into five sections. The first section sets the stage by answering a simple but important question, which is why do design patterns matter in the first place? The next three sections dive into the patterns themselves, following a clear one-chapter, one-pattern approach so you can focus on understanding each idea without distraction. The final section steps back to discuss design principles and common anti-patterns, because knowing what not to build is just as important as knowing what to build.

Almost every functional software system today relies on one or more of these patterns. They quietly shape the architecture of applications across industries and scales. This book focuses on some of the most widely used object-oriented design patterns and demonstrates their implementation using Java. The goal is not to overwhelm you with theory, but to help you understand why these patterns exist, what problems they solve, and when they make sense.

Each chapter begins with a practical design problem. Instead of starting with abstract definitions, we begin with situations you are likely to encounter in real-world development. From there, we examine how a particular pattern helps address the challenge. The emphasis throughout is on clarity and intuition. Wherever possible, examples are grounded in realistic scenarios so you can connect the pattern to the kinds of systems you may already be building.

This book is intended for developers who have a strong foundation in software development and some experience working on medium to large-scale systems. If you have worked with multi-module applications, collaborated across teams, or struggled with maintaining evolving codebases, you will find these concepts especially useful. Whether you are looking to strengthen your understanding of design or refine the way you approach complex problems, this book aims to serve as a practical and reliable guide.

Chapter 1: Enlighten Yourself– This chapter sets the foundation for understanding design patterns, why they exist, and how they emerged from recurring software challenges. It then establishes the mindset required to apply patterns thoughtfully rather than mechanically.

Chapter 2: One of a Kind – This chapter explores the real-world scenarios where controlled instantiation matters. The chapter then discusses the implementation choices in Java, common misconceptions, and how misuse can introduce hidden design risks.

Chapter 3: Object Factory – This chapter examines object creation challenges in growing systems and then walks through practical Java implementations and highlights how factories improve flexibility while avoiding tight coupling.

Chapter 4: Delegate Object Construction– This chapter addresses the complexity of constructing large or configurable objects. It also demonstrates clean Java implementations and discusses readability, immutability, and maintainability benefits.

Chapter 5: Recycle and Reuse– This chapter analyzes performance-driven scenarios where object reuse is critical. The chapter also explains the implementation of trade-offs and when pooling improves efficiency versus when it adds unnecessary complexity.

Chapter 6: Prototype– This chapter explores duplication strategies in object creation. It discusses the cloning techniques in Java, deep versus shallow copying, and when prototypes simplify complex instantiation logic.

Chapter 7: Adapter– This chapter focuses on integration challenges between incompatible interfaces and demonstrates practical Java adaptations, further discussing how to extend legacy or third-party systems safely.

Chapter 8: Decorating Objects– This chapter examines extending behavior without modifying existing code, explains practical layering techniques, and compares decorators with inheritance-based approaches.

Chapter 9: The Guardian– This chapter explores controlled access to objects in real-world scenarios, such as caching and security. The chapter further highlights implementation details and common architectural use cases.

Chapter 10: Simplifying the Complexity– This chapter demonstrates simplifying complex subsystems through unified interfaces. The chapter also discusses how this improves maintainability and reduces unnecessary dependencies.

Chapter 11: Bridge– This chapter addresses systems with multiple dimensions of variation. It also explains structuring abstractions and implementations independently to maintain flexibility over time.

Chapter 12: Applying Filters– This chapter explores designing modular filtering logic. It further discusses combining criteria effectively while preserving clarity and maintainability.

Chapter 13: Composite– This chapter covers modeling hierarchical data structures and explains how uniform treatment of objects simplifies recursive and tree-based designs.

Chapter 14: Template– This chapter examines defining algorithm structure while allowing customization. The chapter also highlights controlled extensibility and reuse in Java implementations.

Chapter 15: Keep a Close Eye– This chapter explores event-driven communication between objects. It demonstrates how to implement observer relationships cleanly and avoid tight coupling.

Chapter 16: State and Behaviors– This chapter analyzes managing behavior transitions without excessive conditionals. It also shows how structured state modeling improves clarity and maintainability.

Chapter 17: Executing Commands– This chapter explores encapsulating actions for flexibility and traceability. It further discusses undo functionality, logging, and decoupled execution models.

Chapter 18: Strategy– This chapter examines replacing conditional logic with interchangeable algorithms and demonstrates runtime flexibility and improved design clarity.

Chapter 19: Pipeline– This chapter explores structuring sequential data processing stages. It also highlights modularity, testability, and extensibility in workflow-based systems.

Chapter 20: Beyond Design Patterns– This chapter reflects on responsible pattern usage, anti-patterns, and design principles, and encourages critical thinking over blind application.

Code Bundle and Coloured Images

Please follow the link to download the
Code Bundle and the ***Coloured Images*** of the book:

https://rebrand.ly/lx4hdee

The code bundle for the book is also hosted on GitHub at **https://github.com/bpbpublications/Software-Design-Patterns-for-Java-Developers-2nd-Edition**. In case there's an update to the code, it will be updated on the existing GitHub repository.

We have code bundles from our rich catalogue of books and videos available at **https://github.com/bpbpublications**. Check them out!

Errata

We take immense pride in our work at BPB Publications and follow best practices to ensure the accuracy of our content to provide with an indulging reading experience to our subscribers. Our readers are our mirrors, and we use their inputs to reflect and improve upon human errors, if any, that may have occurred during the publishing processes involved. To let us maintain the quality and help us reach out to any readers who might be having difficulties due to any unforeseen errors, please write to us at :

errata@bpbonline.com

Your support, suggestions and feedbacks are highly appreciated by the BPB Publications' Family.

At **www.bpbonline.com**, you can also read a collection of free technical articles, sign up for a range of free newsletters, and receive exclusive discounts and offers on BPB books and eBooks. You can check our social media handles below:

[illegible]

Get in touch with us at: **business@bpbonline.com** for more details.

Piracy

If you come across any illegal copies of our works in any form on the internet, we would be grateful if you would provide us with the location address or website name. Please contact us at **business@bpbonline.com** with a link to the material.

If you are interested in becoming an author

If there is a topic that you have expertise in, and you are interested in either writing or contributing to a book, please visit **www.bpbonline.com**. We have worked with thousands of developers and tech professionals, just like you, to help them share their insights with the global tech community. You can make a general application, apply for a specific hot topic that we are recruiting an author for, or submit your own idea.

Reviews

Please leave a review. Once you have read and used this book, why not leave a review on the site that you purchased it from? Potential readers can then see and use your unbiased opinion to make purchase decisions. We at BPB can understand what you think about our products, and our authors can see your feedback on their book. Thank you!

For more information about BPB, please visit **www.bpbonline.com**.

Join our Discord space

Join our Discord workspace for latest updates, offers, tech happenings around the world, new releases, and sessions with the authors:

https://discord.bpbonline.com

Table of Contents

1. Enlighten Yourself

- Introduction
- Structure
- Objectives
- Design patterns
- Classification of design patterns
- Creational design patterns
    - *Familiar use cases*
- Structural design patterns
    - *Familiar use cases*
- Behavioral design patterns
    - *Familiar use cases*
- Importance of software design
- Benefits of a good design
- Think beyond methods and classes
- Conclusion
- Questions
    - *Find the odd one out*
    - *Select two correct answers*
- Answers
    - *Find the odd one out*
    - *Select two correct answers*

2. One of a Kind

- Introduction

Structure
Objectives
Simple logger
Singleton
Singleton design pattern
Variations of the singleton design pattern
    Lazy initialization
    Multiple instances of a singleton
    Eager initialization
    Thread-safe singleton
Reflection and singleton
Enum as a singleton
Singleton and serialization
Advantages and drawbacks
Usage
Conclusion
Questions
    Find the odd one out
    Select two correct answers
Answers
    Find the odd one out
    Select two correct answers

3. Object Factory
Introduction
Structure
Objectives
Interacting with interfaces
Building factories
Response factory
    First run of response factory
The simple factory

- Multiple factories
  - *Factory that speaks French*
  - *Response updated*
  - *Second run of response factory*
- Factory method pattern
- Family of objects
  - *Dependency injection*
  - *Response factory’s third run*
- Abstract factory
- Advantages and drawbacks
- Usage
- Conclusion
- Questions
  - *Find the odd one out*
  - *Select two correct answers*
- Answers
  - *Find the odd one out*
  - *Select two correct answers*

4. Delegate Object Construction

- Introduction
- Structure
- Objectives
- Delegating object construction
  - *Representational state*
- Implementing builders
  - *The interface*
  - *Concrete builders*
  - *Deployment manager*
  - *Testing the builders*
- Summary
- Advantages and drawbacks

- Usage
- Conclusion
- Questions
  - *Find the odd one out*
  - *Select two correct answers*
- Answers
  - *Find the odd one out*
  - *Select two correct answers*

5. Recycle and Reuse

- Introduction
- Structure
- Objectives
- Reusable objects
- The object pool design pattern
- Using an object pool
- Object pool implementation
  - *Pool initialization*
  - *Pool destruction*
  - *Acquiring objects*
  - *Releasing objects*
- Varying implementations
  - *Partial initialization*
  - *Object release*
- Advantages and drawbacks
- Usage
- Conclusion
- Questions
  - *Find the odd one out*
  - *Select two correct answers*
- Answers
  - *Find the odd one out*

Select two correct answers

6. **Prototypes**

Introduction

Structure

Objectives

Prototypes

The prototype design pattern

Implementing the prototype pattern

Prototype interface

Prototype implementations

Testing the prototype

Prototype registry

Importance of a registry

Summary

Advantages and drawbacks

Usage

Conclusion

Questions

Find the odd one out

Select two correct answers

Answers

Find the odd one out

Select two correct answers

7. **Adapter**

Introduction

Structure

Objectives

Incompatible interfaces

Adapter design pattern

Building an adapter

    - *Interfaces*
    - *Concrete implementations*
    - *Adapter*
    - *Testing the adapter*
  - Summary
  - Advantages and drawbacks
  - Usage
  - Conclusion
  - Questions
    - *Find the odd one out*
    - *Select two correct answers*
  - Answers
    - *Find the odd one out*
    - *Select two correct answers*

8. Decorating Objects
  - Introduction
  - Structure
  - Objectives
  - Wrappers
  - The decorator design pattern
  - Building a decorator
    - *Interface*
    - *The subject*
    - *Base decorator*
    - *Concrete decorators*
    - *Testing the decorators*
  - Summary
  - Advantages and drawbacks
  - Usage
  - Conclusion
  - Questions

    - *Find the odd one out*
    - *Select two correct answers*
  - Answers
    - *Find the odd one out*
    - *Select two correct answers*

9. **The Guardian**
  - Introduction
  - Structure
  - Objectives
  - Proxies
  - Proxy design pattern
  - Types of proxies
    - *Remote proxy*
    - *Virtual proxy*
    - *Protection proxy*
  - Building a proxy
    - *Interface*
    - *Subject*
    - *Proxy*
    - *Testing the proxy*
  - Summary
  - Advantages and drawbacks
  - Usage
  - Conclusion
  - Questions
    - *Find the odd one out*
    - *Select two correct answers*
  - Answers
    - *Find the odd one out*
    - *Select two correct answers*

10. **Simplifying the Complexity**

Introduction
Structure
Objectives
Facade
Facade design pattern
Building a facade
    All for one
    One for all
Summary
Advantages and drawbacks
Usage
Conclusion
Questions
    Find the odd one out
    Select two correct answers
Answers
    Find the odd one out
    Select two correct answers

11. Bridge
Introduction
Structure
Objectives
Bridge design pattern
Implementing the bridge pattern
    Existing implementations
    Transaction mode
    Transaction
    Concrete transaction modes
    Concrete transactions
    Testing the bridge
Summary

- Advantages and drawbacks
- Usage
- Conclusion
- Questions
    - *Find the odd one out*
    - *Select two correct answers*
- Answers
    - *Find the odd one out*
    - *Select two correct answers*

12. Applying Filters

- Introduction
- Structure
- Objectives
- Filter design pattern
- Implementing the filter pattern
    - *Filter interface*
    - *Simple filters*
    - *Complex filters*
    - *Logical operations*
        - *Logical operation: AND*
        - *Logical operation: OR*
    - *Testing the filters*
- Summary
- Advantages and drawbacks
- Usage
- Conclusion
- Questions
    - *Find the odd one out*
    - *Select two correct answers*
- Answers
    - *Find the odd one out*
    - *Select two correct answers*

13. Composite
- Introduction
- Structure
- Objectives
- Composite design pattern
- Implementing the composite pattern
  - *Product interface*
  - *Derived types*
  - *Testing the composite design pattern*
- Summary
- Advantages and drawbacks
- Usage
- Conclusion
- Questions
  - *Find the odd one out*
  - *Select two correct answers*
- Answers
  - *Find the odd one out*
  - *Select two correct answers*

14. Template
- Introduction
- Structure
- Objectives
- Template
- Template method design pattern
- Building a template method
  - *Template method*
    - *Varying implementations*
  - *Testing the template method pattern*
- Summary
- Advantages and drawbacks

- Usage
- Conclusion
- Questions
  - *Find the odd one out*
  - *Select two correct answers*
- Answers
  - *Find the odd one out*
  - *Select two correct answers*

15. Keep a Close Eye

- Introduction
- Structure
- Objectives
- Observer design pattern
- Implementing observer pattern
  - *Interfaces*
  - *Subject*
  - *Observers*
  - *Testing the observer design pattern*
- Varying implementations
  - *Push mechanism*
  - *Pull mechanism*
- Summary
- Advantages and drawbacks
- Usage
- Conclusion
- Questions
  - *Find the odd one out*
  - *Select two correct answers*
- Answers
  - *Find the odd one out*
  - *Select two correct answers*

16. **State and Behaviors**
- Introduction
- Structure
- Objectives
- The state design pattern
- Implementing the state pattern
  - *The interface*
  - *The concrete implementations*
  - *The stateful object*
  - *Testing the state design pattern*
- Summary
- Advantages and drawbacks
- Usage
- Conclusion
- Questions
  - *Find the odd one out*
  - *Select two correct answers*
- Answers
  - *Find the odd one out*
  - *Select two correct answers*

17. **Executing Commands**
- Introduction
- Structure
- Objectives
- Command design pattern
- Implementing the command pattern
  - *Receiver*
  - *Command*
  - *Invoker*
  - *Client*
- Summary

- Advantages and drawbacks
- Usage
- Conclusion
- Questions
  - *Find the odd one out*
  - *Select two correct answers*
- Answers
  - *Find the odd one out*
  - *Select two correct answers*

18. Strategy

- Introduction
- Structure
- Objectives
- Strategy design pattern
- Implementing the strategy pattern
  - *Strategy interface*
  - *Strategies*
  - *Selection of strategies*
  - *Invoking strategies*
  - *Testing the strategies*
- Summary
- Advantages and drawbacks
- Usage
- Conclusion
- Questions
  - *Find the odd one out*
  - *Select two correct answers*
- Answers
  - *Find the odd one out*
  - *Select two correct answers*

19. Pipeline

Introduction
Structure
Objectives
Pipeline design pattern
Implementing the pipeline pattern
  Task interface
  Concrete tasks
  Constructing a pipeline
  Testing the pipeline
Summary
Advantages and drawbacks
Usage
Conclusion
Questions
  Find the odd one out
  Select two correct answers
Answers
  Find the odd one out
  Select two correct answers

20. Beyond Design Patterns
Introduction
Structure
Objectives
Quick recap
Summary of patterns
  Classification of class versus object patterns
Design principles
  Don’t Repeat Yourself
  Single responsibility
  Dependency inversion
  Encapsulate what varies

- *Open closed*
- *Favor interface over implementation*

Interface segregation

- *Least knowledge*

Anti-patterns

- *Interface for constants*
- *God object*
- *Caching irregularities*
- *Hard code*
- *Boat anchor*
- *Copy and paste*
- *No silver bullets*

Conclusion

Questions

- *Find the odd one out*
- *Select two correct answers*

Answers

- *Find the odd one out*
- *Select two correct answers*

Index

CHAPTER 1

Enlighten Yourself

Introduction

Software development is not only about solving problems but also about doing so in an efficient manner. Problem-solving is a process that involves multiple steps. It starts with problem identification, advances towards developing an algorithm for it, and ends with an implementation. Software can be implemented in many ways; right from devising an algorithm to using a correct set of data objects and the integration among them; all of this is a part of implementing software. However, without a good design, even the best implementations could fall apart.

Designing software is not an easy task to accomplish; it requires a lot of effort and knowledge to come up with a design solution that is easy to implement, feasible, adheres to the programming principles, and is scalable in nature. Nowadays, the amount of data a software processes is not only greater in terms of storage but also much more complex. With so much information being kept in the software systems, it becomes important to inspect the current design and transition to a better one that will adapt to increased usage and fast processing requirements. Software design is not only about putting elements in place in a defined manner, but it is also about establishing interconnected pathways that help to transfer data among multiple controllable structures to form a flow that, when executed, efficiently performs a task.

To put it interestingly, the design of the software gives it life. It not only supports the underlying framework but also makes it easy for the developers to scale the system and be ready for future changes, even when, at times, they are unaware of them. A good design ensures that the software is robust and can adapt to new features easily and with less time than is possibly required when there is no proper design.

Structure

This chapter covers the following topics:

- Design patterns
- Classification of design patterns
- Creational design patterns
- Structural design patterns
- Behavioral design patterns
- Importance of software design
- Benefits of a good design
- Think beyond methods and classes

Objectives

By the end of this chapter, the readers will develop a thorough understanding of software design patterns and their role in software development. They will explore the different categories of design patterns along with their importance and benefits, and will learn about the benefits of a good design and the use of design patterns in software development.

After finishing this chapter, the readers will be able to understand the basics of design patterns and will be able to classify them into categories. They will also be able to distinguish between the underlying approaches that form the basis of categorization.

Design patterns

Design patterns, in software engineering, are the blueprints or templates that emerge out of well-designed exercises to solve typical problems that are usually recurring in nature. They can also be considered as the best practices employed to effectively overcome software design problems.

Design patterns follow a systematic approach and can be reused in a variety of situations as they provide us with an idea of how to solve a particular problem. They are not the actual code solutions, but instead blueprints or general solutions that help us to solve the problem effectively.

A design pattern can have one or more standard objects or interfaces that are necessary for its successful implementation. These interfaces or objects are the very soul of the pattern itself and require the programmer to follow the exact practice as described in the pattern for effective utilization.

Figure 1.1 exhibits some of the core attributes of design patterns:

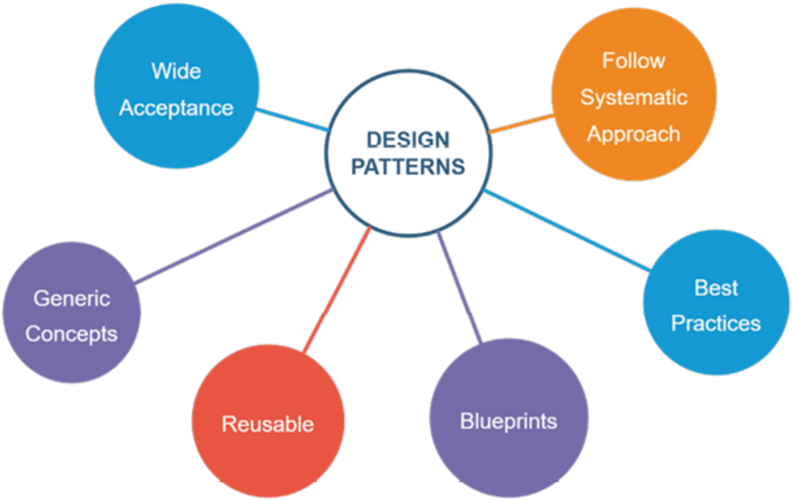


***Figure 1.1**: Design patterns*

Classification of design patterns

The design patterns help solve a variety of design problems ranging from object creation and interface development to managing the behavioral aspects of an object. They address the design problems and make it easier to reach solutions that are easy to implement and manage.

An important aspect of software design is the knowledge and understanding of the person who designs it. It is therefore not just recommended but necessary to understand the expected application behavior before selecting a design pattern. While knowing which design pattern is the correct choice, it is equally important to know which ones are not. The implication of selecting the wrong pattern can be far more troublesome than expected.

Figure 1.2 exhibits the classification of design patterns into three categories, that is, creational, structural, and behavioral:

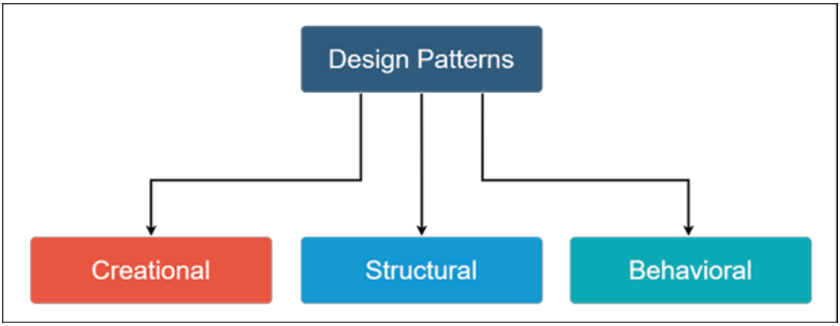


***Figure 1.2**: Design patterns classification*

While the creational patterns are more involved with the creation of objects and the interfaces that connect with those objects, the structural patterns help with designing large structures by assembling two or more systems to provide new functionalities, and the behavioral patterns are inclined towards communication between objects.

Creational design patterns

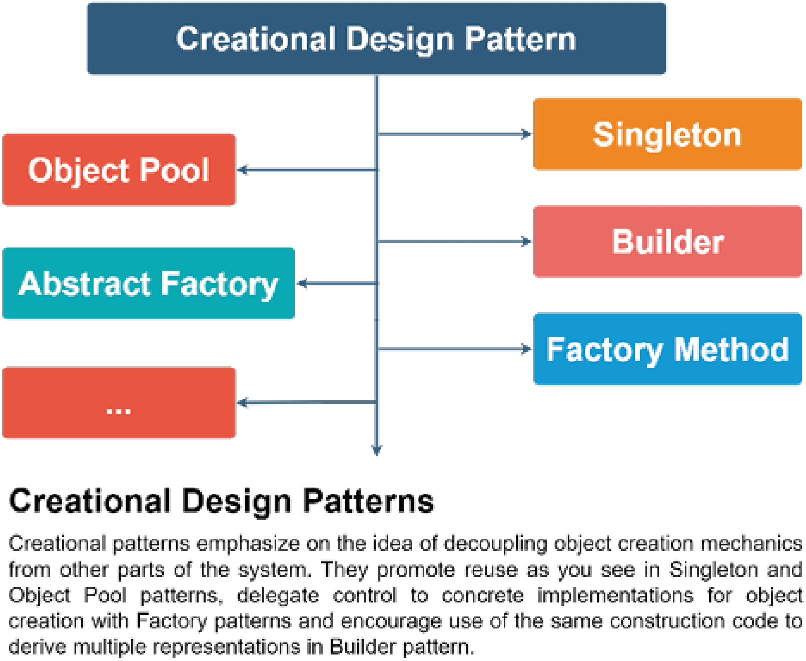
Design patterns that relate to the idea of object creation in an efficient and controlled manner are referred to as creational design patterns.

Their work could be understood based on two defining principles:

- Hiding the object and its composition from the outside world
- Controlling object usage

Both principles are followed by providing an abstraction layer that ensures the only connection to these objects from the outside world is through this layer.

Figure 1.3 explains the objective of creational design patterns and lists some of the patterns discussed in this book:



Creational Design Patterns

Creational patterns emphasize on the idea of decoupling object creation mechanics from other parts of the system. They promote reuse as you see in Singleton and Object Pool patterns, delegate control to concrete implementations for object creation with Factory patterns and encourage use of the same construction code to derive multiple representations in Builder pattern.

***Figure 1.3**: Creational design pattern*

Creating objects and using them without proper access controls makes it increasingly difficult to manage and refactor the application in the future. The creational design patterns decouple the process of object creation from its usage and ensure that the objects are created and accessed in a controlled manner.

Creational design patterns make effective use of abstraction and access control mechanisms provided by object-oriented languages. Interfaces are used to reduce the dependency of objects on each other, as we see in the factory pattern. Access control mechanisms help to control access to an object and its state directly by any other object.

The following list outlines the essential learning to remember about the creational design patterns:

- They segregate object creation from other business logic.
- They promote reusability of instances.
- They control object access.
- They hide object composition.

In this book, we will discuss some of the widely used and accepted creational design patterns, some of which are as follows:

- **Singleton design pattern**: This pattern ensures that only one instance of the class is exposed for access to the entire application per Java **Virtual Machine** (**VM**).
- **Builder design pattern**: This pattern aims to simplify the construction process of complex objects by providing a stepwise approach to instance creation. It also exhibits the capability to construct the same object with different representations.
- **Object pool design pattern**: This pattern creates a pool or group of objects that can be reused to process some information. It is used in scenarios where the object creation cost is high, and object reuse is more advisable than creating a new one.
- **Factory design pattern**: This pattern provides an abstraction and delegates the control to the subclasses for the type of object to be constructed. It is the responsibility of the child classes to decide upon the type of object based on the supplied input(s).

Familiar use cases

The following list outlines some of the common use cases where creational design patterns can be useful:

- Imagine an object that acts as a cache and is kept in sync with the database. Having multiple such objects can lead to inconsistencies and corrupted data, leading to incorrect computations; mainly because reads and writes to the cached data are stored in more than one place. By using the singleton design pattern, we can avoid all of that by ensuring that there is only one cache object that needs to be kept in sync with the database. An example of a singleton in the Java Development Kit is the class runtime, which has only one instance in a JVM.

- Imagine a scenario where multiple objects with a high cost of creation are required to perform parallel or concurrent operations. This scenario is an excellent use case for the object pool design pattern. An object pool instantiates and stores multiple objects that can be reused in an application that has to process multiple requests at a time. An object pool lends its resources for processing and computation and usually has a bounded queue to keep track of the unprocessed requests that wait for their turn until one of the objects in the pool becomes available. An example of an object pool in the JDK is the Executor framework.
- Imagine that you need to calculate tax for a user, and the tax calculation algorithm involves multiple factors. Using the factory design pattern, the calculation process could be simplified and streamlined by introducing these factors into the selection process to select the right implementation.

Structural design patterns

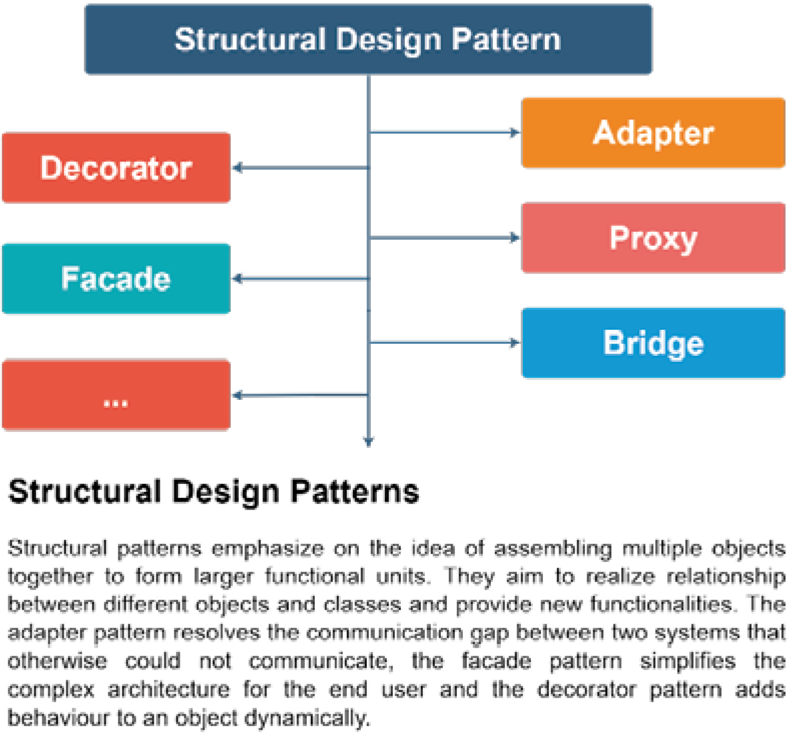
The basic idea behind structural design patterns is the formation of large structures by assembling various objects or their classes to provide new functionalities.

Structural design patterns do not modify existing classes or objects but instead add new features on top of them. This way, in general terms, multiple business use cases can be solved effectively with minor or no architectural changes.

In other words, structural design patterns aim for the realization of relationships among objects and provide new functionalities in return. They make use of the two important pillars of object-oriented programming, that is, inheritance and composition, to form the new structures. The emphasis is on the use of interfaces rather than concrete implementations so that the participating classes remain loosely coupled to each other. This brings us to the important design principles, that is, interact with interfaces rather than with concrete implementations.

Though the larger structure acts as one unified unit, the participating classes and objects can be improved, maintained, and replaced in isolation. Structural design patterns make use of existing classes and objects, wrap them, or use them with previously unrelated or newly formed classes to create new functionalities.

Figure 1.4 explains the objective of structural design patterns and lists some of the patterns discussed in this book:



Structural Design Patterns

Structural patterns emphasize on the idea of assembling multiple objects together to form larger functional units. They aim to realize relationship between different objects and classes and provide new functionalities. The adapter pattern resolves the communication gap between two systems that otherwise could not communicate, the facade pattern simplifies the complex architecture for the end user and the decorator pattern adds behaviour to an object dynamically.

***Figure 1.4**: Structural design pattern*

Facts to remember about the structural design patterns are as follows:

- They assemble classes and objects to form large structures.
- They express the relationship between objects.
- They promote inheritance and composition.

- They require no or minimal change in participating classes.
- They realize new functionality.

In this book, we will discuss some of the widely used and accepted structural design patterns, some of which are mentioned in the following list:

- **Adapter design pattern**: It converts one interface into another and provides a way for classes to interact with each other without changing their source code.
- **Decorator design pattern**: It expands the functionality of an existing class or interface without modifying its source code. It creates wrappers that inherit the same interface and use composition to add the new functionality.
- **Proxy design pattern**: It acts as a placeholder for another class. The proxy pattern is used as a gateway to control access to other classes. It could be used for security, validation, data retrieval, and other requirements that need controlled access to an object.
- **Facade design pattern**: It simplifies the interaction between two systems by providing an interface that hides the complex processing and data manipulation.

Familiar use cases

Some of the common use cases where structural design patterns can be useful are as follows:

- Imagine a scenario where a ticket booking module confirms the booking only after receiving a response from the payment processing module. These two modules, though unrelated, are supposed to work together to ensure the correct functionality of the overall system. The structure of the data and the tables could be different for both modules, and might require formatting or conversion of data before it is transferred between these two modules. The adapter design pattern is suitable for this use case and requires minimal or no change to the existing modules.
- Think of a system that uses extensive logging, and the existing classes are closed for modification, but a new

requirement needs additional logs. In this scenario, you can make use of the decorator design pattern by creating a wrapper class that has an instance of the log writer class. Now, instead of calling the existing log writer, you can call the instance of the wrapper class that delegates the call to the old wrapper class after writing the additional logs. The file handling capabilities in JDK make use of the decorator design pattern.

- Think of an image search and rendering system that searches its large database to provide relevant images based on the input criteria. The system might be overwhelmed by the increase in the number of searches, and its performance could be reduced.

The proxy design pattern could be utilized to safeguard the system with minimal impact on the user experience. However, by using this pattern, a pre-defined small set of cached images could be returned in response to the search request, and a database request for the actual search could be triggered in the background for rendering additional images to the user.

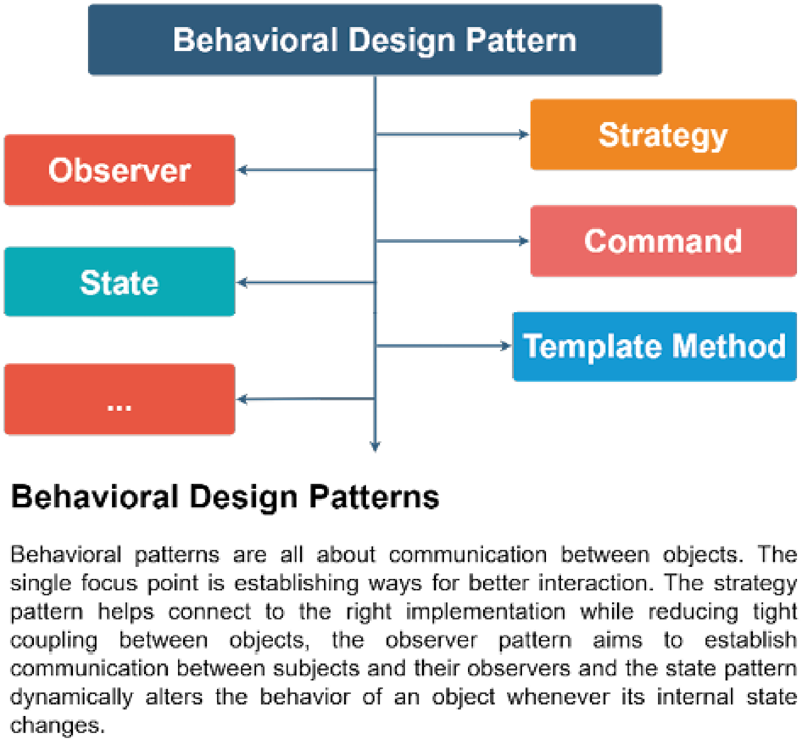
Behavioral design patterns

Behavioral design patterns are all about the relationship between objects and how they interact with each other. They aim to reduce the communication complexity and help improve the interaction among objects. The crux of the design solutions provided by the behavioral design patterns is the use of abstraction to decouple the classes from each other and ensure smoother communication via interfaces.

The behavioral design patterns evaluate the communication process followed within a system and transform it into a form that helps streamline the communication mechanism.

A system's behavior is often dependent on internal factors and external inputs. Behavioral design patterns provide solutions that aid and ease the selection of appropriate behavior or implementation while ensuring reliability and correctness.

Figure 1.5 explains the objective of creational design patterns and lists some of these patterns that we will learn in this book:



Behavioral Design Patterns

Behavioral patterns are all about communication between objects. The single focus point is establishing ways for better interaction. The strategy pattern helps connect to the right implementation while reducing tight coupling between objects, the observer pattern aims to establish communication between subjects and their observers and the state pattern dynamically alters the behavior of an object whenever its internal state changes.

Figure 1.5*: Behavioral design pattern*

The basic idea behind behavioral design patterns can be summed up as:

- Evaluation of the communication mechanism to realize it as a pattern.
- Use of interfaces to support communication among objects.

The following list highlights the facts to remember about the behavioral design patterns:

- They focus on improving communication among classes and objects.
- They promote interaction via interfaces.
- They support multiple behaviors and supervise their selection process.

In this book, we will discuss some of the widely used and accepted structural design patterns, some of which are mentioned in the following list:

- **Strategy design pattern**: It simplifies the selection process among multiple implementations by providing a common interface that all the implementations must respect.
- **Observer design pattern**: It streamlines the communication between a subject and its observers. It relies on the push mechanism of data transfer, and any change in the state of the subject is notified to the observers in near real-time.
- **State design pattern**: It helps to maintain the internal states of an object and allows for a seamless change in the object's behavior with the state transition.
- **Command design pattern**: It encapsulates a request in the form of an object that could be invoked when required. The command can be parameterized, logged, or queued for timed execution.

Familiar use cases

Some of the common use cases where behavioral design patterns can be useful are mentioned in the following list:

- Imagine an application that provides a facility to translate between multiple languages. It can have several implementations that can be triggered based on a set of input variables. For the process of translation to be effective and error-free, it is necessary to select the correct implementation. This particular use case is well-suited for strategy design patterns. The implementations here are the strategies, and only one of them should be followed to obtain the correct result. By using the strategy design pattern, we can develop a centralized logic that can be utilized to decide the correct implementation based on the inputs provided.
- Imagine a stock trading system that keeps track of the current value of all the registered stocks. This information is very important to the stockbrokers who buy and sell stocks regularly. If this information could be delivered to the

brokers in near real-time, it would help them immensely.

One of the ways to establish such a mechanism is to expose **application programming interfaces** (**APIs**) that are consumed by the software used by the brokers. Another way to deliver this important information to all the brokers is by introducing the observer design pattern. The observer pattern facilitates the information delivery process by registering all the interested broker clients to the stock trading system and broadcasting changes to the stock prices to all the clients whenever such a change happens. Java JDK also exposes observer and subject interfaces to realize the observer design pattern.

- If you have ever used the kiosk that prints boarding passes for flights at the airport, you would have noticed that it follows a series of steps to process your information and print the boarding pass. You would also have noticed that the next button not only takes you to the next step but also processes your information in multiple ways at different steps. Those steps can be imagined as the states of the software running on those kiosks. By using the state design pattern, the press of a button would imply a change in the internal state of the kiosk, subsequently changing its behavior with every state change.

Importance of software design

Sometimes, it is very difficult to make changes to the existing codebase that even the smallest of the changes require modification(s) in multiple classes, and a change in one part of the system may break something else. At times, the performance is so undesirable that the need for a new system is observed. All these questions often end with the same answer, and the answer is bad design choices.

The planning and design phase is of utmost importance for most of the preceding discussion. Designing a new system or adding a new feature to the existing one comes with a lot of challenges, especially if the system's complexity is high. A bad design choice could end up in a low-performance and unhealthy system that could further introduce a lot of bugs and long working hours to fix them.

A properly designed system exhibits many good properties that increase the functional and performance capabilities of the system. A good design streamlines the architecture and reduces coupling between modules. It helps the system to scale with time, perform better under stress, and promote plug-and-play. A good design is not only relevant today, but in the future as well. Having a good design ensures that the system is scalable and can adapt to changes easily.

Benefits of a good design

A good design streamlines the codebase, decongests the system, reduces tight coupling, centralizes control structure, and improves performance and readability. The following list outlines some of the benefits of a good design:

- **Coupling**: When two or more classes are interdependent and even a small change in one affects the other, they are termed tightly coupled. A good design promotes interaction via interfaces and introduces abstraction so that the classes can be modified independently without affecting each other.
- **Readability**: To understand the code, the code must be readable, that is, you should be able to grasp the logic without much difficulty and in as little time as possible. A good design promotes readability by simplifying the architecture.
- **Centralization**: It is favored to keep the important and reusable logic, for example, object access, creation, etc., in a centralized location. This allows for quick modification and system-wide reflection. It is difficult to manage when the same logic is scattered throughout the application, as the slightest change will have to be made in all places without fail.
- **Streamlines codebase**: One of the most important characteristics of a good design is that it should ease the movement of data and remove unnecessary hops.
- **Performance**: By streamlining the communication

mechanism and reducing the number of hops, the design patterns help to improve the performance of a system. A good design also reduces the possibility of architectural bugs that could be hard to trace and mitigate.

Figure 1.6 exhibits the benefits of a good design:

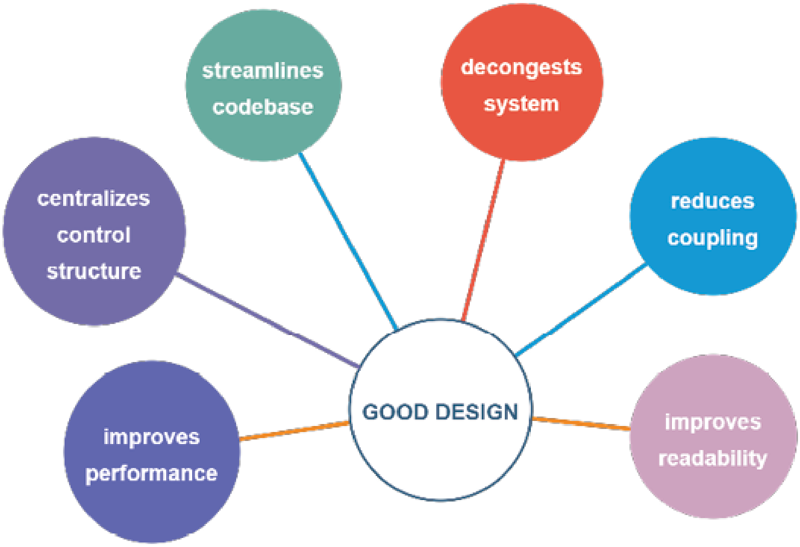

streamlines codebase
decongests system
centralizes control structure
reduces coupling
GOOD DESIGN
improves performance
improves readability

***Figure 1.6**: Benefits of good design*

Think beyond methods and classes

Most of the facts discussed in this chapter revolve around the basic building blocks of a software application, that is, classes and objects, and the various concepts that bind them together. While it is important to understand the concept of design patterns in the world of these building blocks, it is also important to understand and give due attention to the overall architecture of an application.

While designing a system, one cannot simply go into the details of implementation before analyzing the various use cases and capturing the requirements the system is supposed to fulfill. This activity provides a visual description of how a system should behave and the functionalities that should be built.

The following exercise will generate a holistic view of the system and how it could be broken down into multiple modules that interact with each other to form the complete system. The development of multiple modules, each responsible for a different task, presents us with the opportunity to design them separately, selecting the best design patterns that perfectly fit the use case solved by that module.

Though the connection between classes could be a trivial thing to establish, it is imperative to define these connections relevant to the flow of data and the purpose at hand. The design patterns help us with this definition by showcasing various well-tested and widely accepted best practices to choose from.

Refer to *Figure 1.7* and find the perfect match for the following with the appropriate corresponding options:

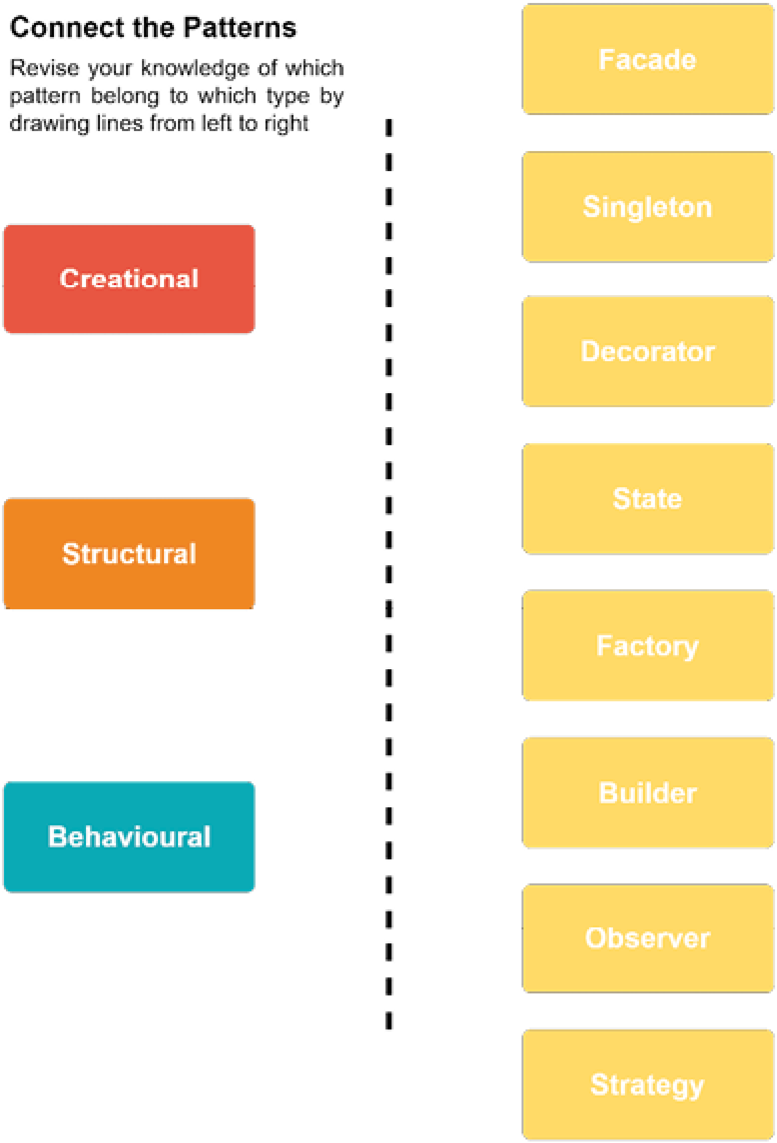

Connect the Patterns
Revise your knowledge of which pattern belong to which type by drawing lines from left to right
Creational
Structural
Behavioural
Facade
Singleton
Decorator
State
Factory
Builder
Observer
Strategy

Figure 1.7: Match the following

The following are the answers of perfect match:

1. **Creational**: Singleton, factory, builder
2. **Structural**: Facade, decorator
3. **Behavioural**: State, observer, strategy

Conclusion

In this chapter, we learned that software development is a subject of continuous evolution. With ever-changing technology and ground-breaking innovations, there will always be a need for better design and organization that facilitates and improves the development process.

We also discussed the basics of software design and briefly explored various patterns that can be utilized to improve the process of software development. We also discussed some of the use cases where those patterns could be used and benefit from.

In the next chapter, we will build a thorough and comprehensive understanding of some of these design patterns. The chapters will be bundled into three different sections, one each for creational, structural, and behavioral design patterns. The chapters will take you on a tour of various patterns, discussing not just the basics but also the defining principles on which those patterns are based. There will be coding examples to guide you throughout the chapter and to ensure that you are ready to utilize the pattern as soon as you finish the chapter.

Questions

In this section, we will sharpen our brains by indulging in some fun brain activities. Let us get started.

Find the odd one out

1. Singleton, facade, factory method, abstract factory
2. State, proxy, strategy, observer
3. Proxy, decorator, adapter, singleton

Select two correct answers

1. Structural design pattern promotes:
    a. Minimal change in existing classes
    b. Inheritance and composition
    c. Rewriting classes to support new functionality

    - d. Assembling existing structures to form new ones
2. Behavioral design pattern focuses on:
    - a. Improving communication
    - b. Interaction via interfaces
    - c. Tight coupling
    - d. Support for multiple behaviors
3. Loose coupling is achieved by:
    - a. Grouping classes together
    - b. Introducing abstraction in the form of interfaces
    - c. Directly calling an object from another
    - d. Creating singleton objects
4. What is composition?
    - a. When an object has a reference to another object
    - b. When a class inherits another class
    - c. When a method has multiple lines
    - d. Building complex types by combining simpler objects
5. What is the benefit of using interfaces?
    - a. They help in writing better code
    - b. They provide default-level access control
    - c. They promote loose coupling
    - d. They enable multiple implementations of behavior
6. Three things that a good design pattern provides:
    - a. Readability
    - b. Reusability
    - c. Tight coupling
    - d. Centralization

Answers

This section provides the answers to the questions given.

Find the odd one out

1. Facade
2. Proxy
3. Singleton

Select two correct answers

1. Inheritance and composition, assembling existing structures to form new ones
2. Improving communication, support for multiple behaviors
3. Introducing abstraction in the form of interfaces, using dependency injection
4. When an object has a reference to another object, building complex types by combining simpler objects
5. They promote loose coupling, and they enable multiple implementations of behavior
6. Readability, Reusability

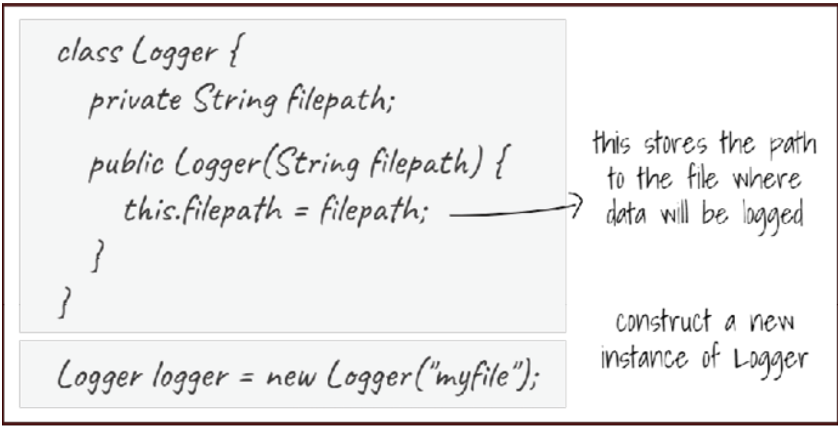
CHAPTER 2

One of a Kind

Introduction

Objects are the centrepiece of any object-oriented programming language; they are the elements on which the whole application stands. In this chapter, we will discuss a very special object, a singleton, and will learn about the singleton design pattern. Singletons are unique objects; there is only one instance of their type in the whole application. Object creation is often a trivial task; for example, one can make use of the new keyword to construct a new object.

In *Figure 2.1*, the class logs the information to a file. It contains a public constructor and a data member that stores the path to the file where the data will be logged:



```
class Logger {
  private String filepath;

  public Logger(String filepath) {
    this.filepath = filepath;
  }
}
```

```
Logger logger = new Logger("myfile");
```

this stores the path to the file where data will be logged
construct a new instance of Logger

***Figure 2.1**: Introduction*

Structure

This chapter covers the following topics:

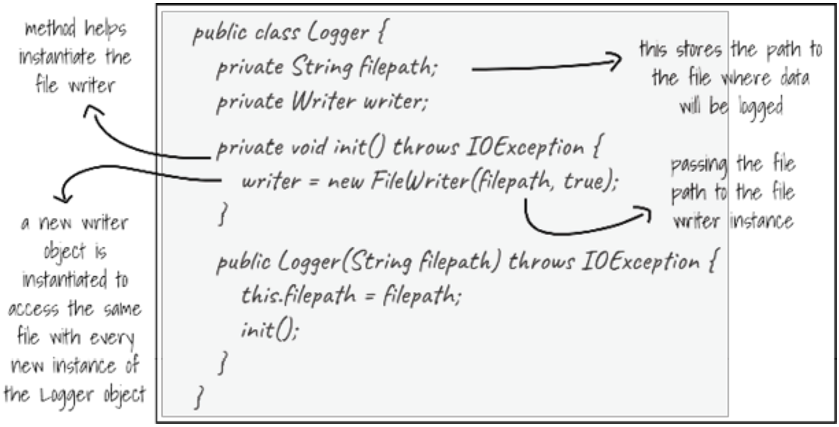
- Simple logger
- Singletons
- Singleton design pattern
- Variations of the singleton design pattern
- Lazy initialization
- Multiple instances of a singleton
- Eager initialization
- Thread-safe singletons
- Reflection and singleton
- Enum as a singleton
- Singleton and serialization
- Advantages and drawbacks
- Usage

Objectives

By the end of this chapter, the readers will be able to understand the basics of a singleton object. They will learn about the singleton design pattern and the different variations of it. Readers will also learn about the challenges of using serialization with a singleton, the effects of reflection on singleton objects, and how to avoid them. They will understand the implementation details of the many variations of the singleton design pattern, the benefits it provides, and a few drawbacks, along with some of the places where the singleton design pattern can be used.

Simple logger

In *Figure 2.1*, the class did not have any mechanism to log information. Let us add some methods to make our class realistic and useful. *Figure 2.2* exhibits an updated version of the logger class that uses a writer to write to the log files:



```
public class Logger {
  private String filepath;
  private Writer writer;

  private void init() throws IOException {
    writer = new FileWriter(filepath, true);
  }

  public Logger(String filepath) throws IOException {
    this.filepath = filepath;
    init();
  }
}
```

method helps instantiate the file writer

this stores the path to the file where data will be logged

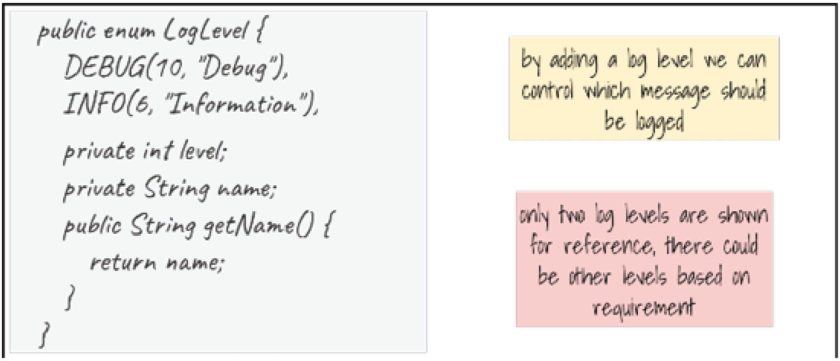
passing the file path to the file writer instance

a new writer object is instantiated to access the same file with every new instance of the Logger object

***Figure 2.2**: Logger*

The class looks more practical now; it initializes a writer object to write the data. Let us add some more methods to our class and create an enum that will help to segregate the logged information. Segregating logs is a good way to manage and search for the required information.

By using a log level, messages can be logged based on their level of severity or importance, as shown in *Figure 2.3*:



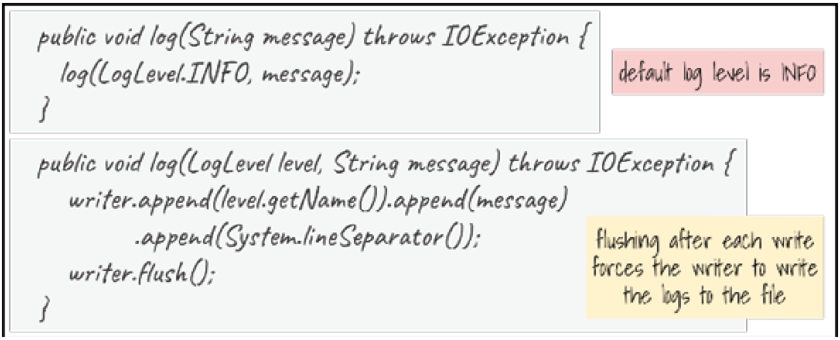
```
public enum LogLevel {
  DEBUG(10, "Debug"),
  INFO(6, "Information"),
  private int level;
  private String name;
  public String getName() {
    return name;
  }
}
```

by adding a log level we can control which message should be logged

only two log levels are shown for reference, there could be other levels based on requirement

***Figure 2.3**: Log level*

Figure 2.4 exhibits the methods for logging information to the files:



```
public void log(String message) throws IOException {
  log(LogLevel.INFO, message);
}

public void log(LogLevel level, String message) throws IOException {
    writer.append(level.getName()).append(message)
          .append(System.lineSeparator());
    writer.flush();
}
```

default log level is INFO

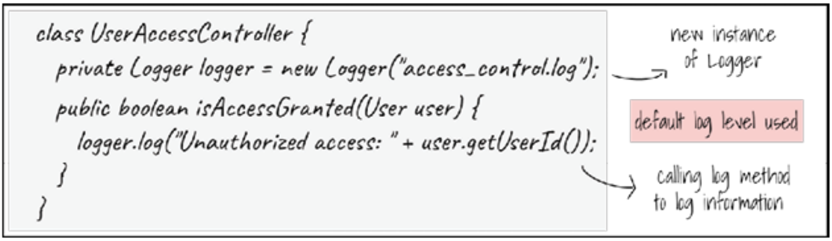
flushing after each write forces the writer to write the logs to the file

***Figure 2.4**: Adding log level to logger*

These two methods complete the implementation of the logger class demonstrated in *Figure 2.2*.

If every class has its instance of logger, there will be as many file writers as there are logger objects, connecting to and accessing the same log file concurrently, leading to possible performance degradation.

In *Figure 2.5*, we have created a simple controller that has an instance of logger to write to a file of our choice. It uses the default log method demonstrated in *Figure 2.4*.



```
class UserAccessController {
  private Logger logger = new Logger("access_control.log");
  public boolean isAccessGranted(User user) {
    logger.log("Unauthorized access: " + user.getUserId());
  }
}
```

new instance of Logger

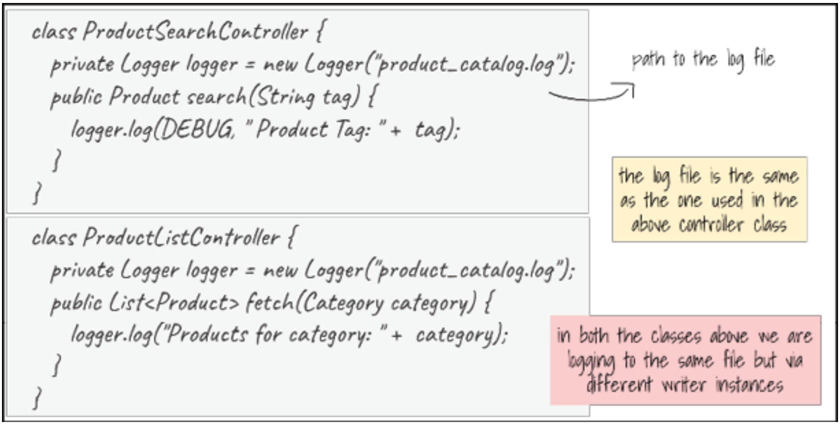
default log level used

calling log method to log information

***Figure 2.5**: Using logger*

Imagine every class opening separate connection(s) to the file(s), soon the system will have a huge number of connections to manage. Not only does it involve a lot of unwanted management, but it also impacts the performance of the system. If we could have a fixed set of connections for the whole application to use, the situation could be improved.

In *Figure 2.6*, although we are logging into the same log file, we have two different instances of logger, and both open separate connections to the log file:



```
class ProductSearchController {
  private Logger logger = new Logger("product_catalog.log");
  public Product search(String tag) {
    logger.log(DEBUG, " Product Tag: " + tag);
  }
}
class ProductListController {
  private Logger logger = new Logger("product_catalog.log");
  public List<Product> fetch(Category category) {
    logger.log("Products for category: " + category);
  }
}
```

path to the log file

the log file is the same as the one used in the above controller class

in both the classes above we are logging to the same file but via different writer instances

***Figure 2.6**: Controllers*

Singleton

A singleton has only one instance throughout the application. A singleton, by design, is global and can be accessed from anywhere in the application by using the exposed accessor. The accessor is designed to return the same instance of the class, no matter how many times it is called.

A singleton is useful in scenarios where only a single instance of a type is required throughout the application, making it the single point of access for that type. A singleton is often confused with global access. It should be kept in mind that for an object to be global, it is not required to be a singleton.

A singleton also provides some interesting add-ons; since a singleton is the single point of access for a type, it could also be utilized to restrict access to it or to provide partial access based on certain configurations.

Singleton design pattern

The singleton design pattern is one of the simplest to learn and understand. It is also one of the most used design patterns.

Figure 2.7 exhibits the structure of a singleton object:

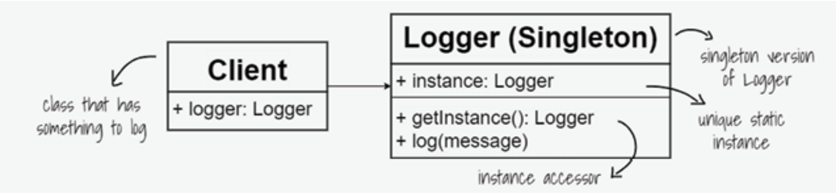
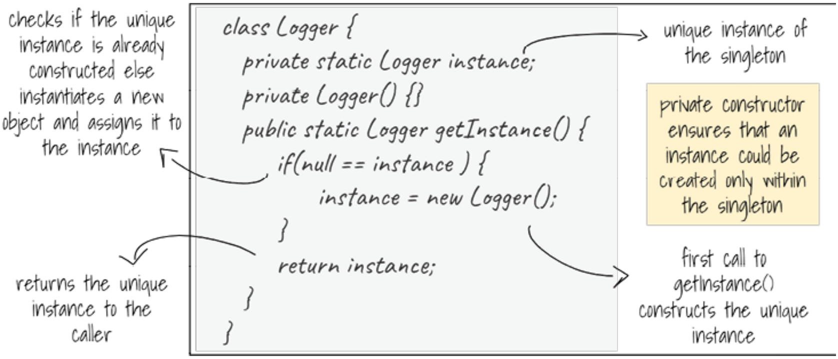

class that has something to log
Client
+ logger: Logger
Logger (Singleton)
+ instance: Logger
+ getInstance(): Logger
+ log(message)
singleton version of Logger
unique static instance
instance accessor

***Figure 2.7**: Singleton design pattern*

Figure 2.8 exhibits the logger class as a singleton, with a static instance of itself that is accessible by a static method. The accessor method takes care of instantiating the instance and returns it to the caller.



```
class Logger {
  private static Logger instance;
  private Logger() {}
  public static Logger getInstance() {
    if(null == instance ) {
      instance = new Logger();
    }
    return instance;
  }
}
```

checks if the unique instance is already constructed else instantiates a new object and assigns it to the instance
returns the unique instance to the caller
unique instance of the singleton
private constructor ensures that an instance could be created only within the singleton
first call to getInstance() constructs the unique instance

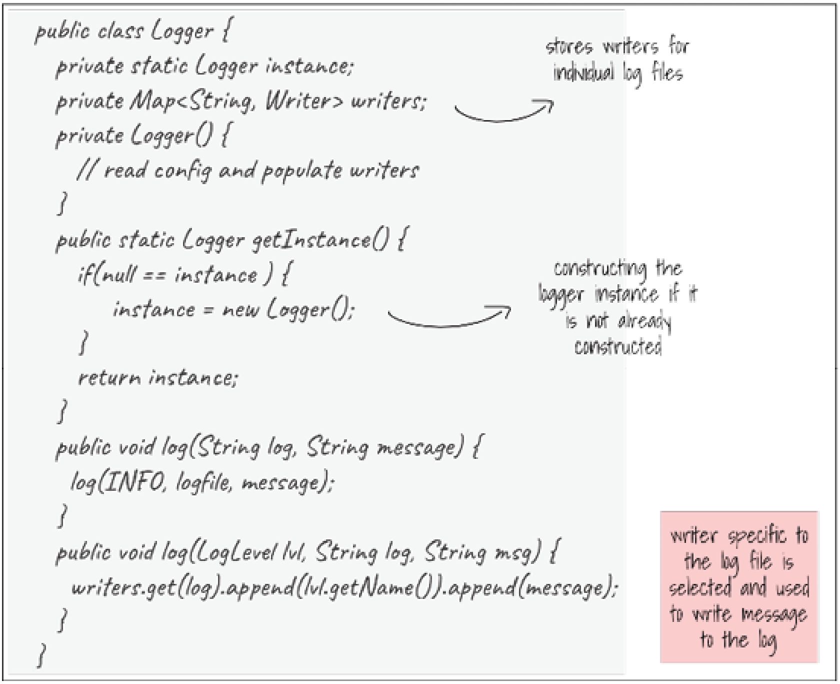
***Figure 2.8**: Update*

Instead of instantiating a new instance of the logger in each distinct class, the same instance can now be reused. The example is kept simple to demonstrate and focus on the application of the singleton.

The approach is to provide a configuration file that is read before constructing the singleton object. The configuration provides the information related to the log files that the application will use, writer objects for all these log files are constructed, and their references are kept in a map to support logging operations.

Let us make changes to our existing controller class to utilize our new singleton object and log some messages. The redesigned **Logger** class is a small and effective solution for immediate logging requirements, but it is not the only one.

Figure 2.9 exhibits the complete implementation of the **Logger** class with a map of writers and methods to log information:



```
public class Logger {
  private static Logger instance;
  private Map<String, Writer> writers;
  private Logger() {
    // read config and populate writers
  }
  public static Logger getInstance() {
    if(null == instance ) {
        instance = new Logger();
    }
    return instance;
  }
  public void log(String log, String message) {
   log(INFO, logfile, message);
  }
  public void log(LogLevel lvl, String log, String msg) {
   writers.get(log).append(lvl.getName()).append(message);
  }
}
```

stores writers for individual log files

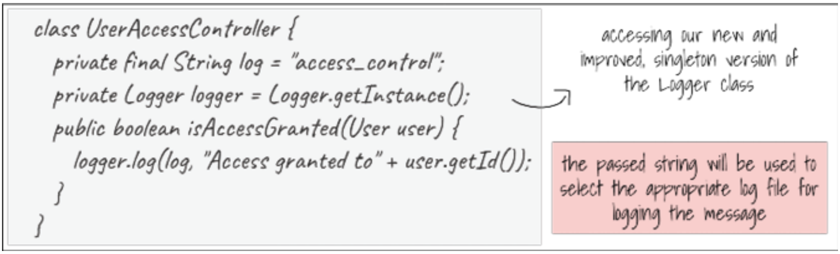
constructing the logger instance if it is not already constructed

writer specific to the log file is selected and used to write message to the log

***Figure 2.9**: Logger as a singleton*

Our updated implementation of the **Logger** class can now be used to log information to the file path, and it is passed as input to the log methods.

Figure 2.10 exhibits the use of the updated **Logger** class from another class:



```
class UserAccessController {
  private final String log = "access_control";
  private Logger logger = Logger.getInstance();
  public boolean isAccessGranted(User user) {
    logger.log(log, "Access granted to" + user.getId());
  }
}
```

accessing our new and improved, singleton version of the Logger class

the passed string will be used to select the appropriate log file for logging the message

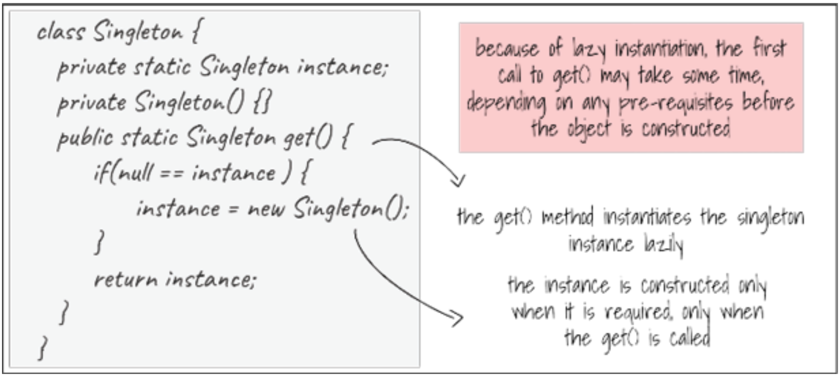
***Figure 2.10**: User access controller*

Variations of the singleton design pattern

There are many ways in which a singleton object can be constructed. Although the basic design remains the same, the algorithm could be modified to utilize mechanisms such as eager and lazy initialization, double-check locking, and even enums.

Lazy initialization

The original implementation of a singleton utilizes lazy initialization as described in the preceding example of the **Logger** class. Refer to *Figure 2.11*:



```
class Singleton {
  private static Singleton instance;
  private Singleton() {}
  public static Singleton get() {
      if(null == instance ) {
          instance = new Singleton();
      }
      return instance;
  }
}
```

because of lazy instantiation, the first call to get() may take some time, depending on any pre-requisites before the object is constructed

the get() method instantiates the singleton instance lazily

the instance is constructed only when it is required, only when the get() is called

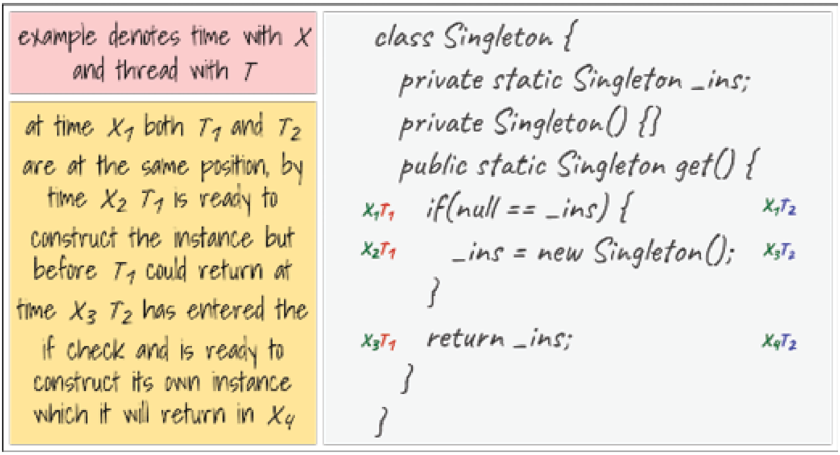
***Figure 2.11**: Lazy singleton*

With this implementation, the singleton instances are constructed only when they are required, i.e., when the first call to access the singleton is made. Since the instance is constructed lazily, the instantiation process could take some time, and the end user might experience some delay in object construction. This delay, however, is only for the first time, and the instance would be immediately returned in the subsequent calls.

Multiple instances of a singleton

The preceding example works well in a single-threaded environment but is not suitable for a multithreaded one. With a single thread accessing the singleton object, there is no reason for multiple instances of the same *singleton* object to be accidentally constructed. When multiple threads are active, two or more threads can simultaneously access and construct multiple instances, breaking the concept of a singleton object and returning different objects of that kind instead of one as it should be.

Think of a singleton that counts the number of requests it received for future analysis, apart from doing its primary job. If there were more than one instance of this singleton, the data would be incorrect and so would the analysis. In *Figure 2.12*, multiple threads are accessing the singleton simultaneously:



example denotes time with X and thread with T

at time X_1 both T_1 and T_2 are at the same position, by time X_2 T_1 is ready to construct the instance but before T_1 could return at time X_3 T_2 has entered the if check and is ready to construct its own instance which it will return in X_4

```
class Singleton {
  private static Singleton _ins;
  private Singleton() {}
  public static Singleton get() {
    if(null == _ins) {
      _ins = new Singleton();
    }
    return _ins;
  }
}
```

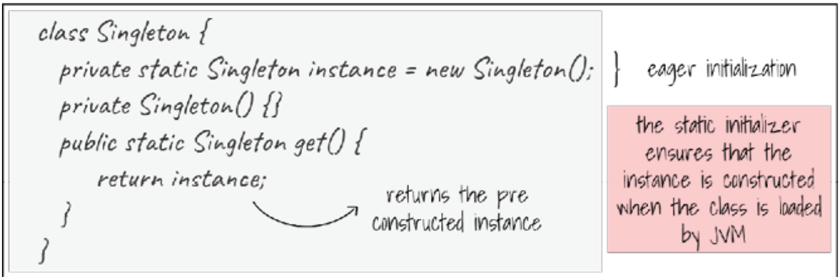
Figure 2.12: *Multiple instances of a singleton*

For any multithreaded application, in the absence of a mechanism to restrict multiple threads from accessing the same block of code simultaneously, multiple threads can access that same block at a single time.

In order to conform to the principles of the singleton and to ensure that multiple instances of a singleton object cannot be instantiated within the same application, other variations of the singleton design patterns can be employed. There are multiple ways to construct a singleton object, and all of them provide a different flavor to the construction process.

Eager initialization

With eager initialization, the object is constructed when the class is loaded by the Java class loader. The eager initialization process rectifies the multiple instance problem since there is no way for more than one thread in an application to access the construction logic simultaneously. This is made possible because of a static initializer that ensures the construction of instances when the class is loaded by the JVM, as shown in *Figure 2.13*:



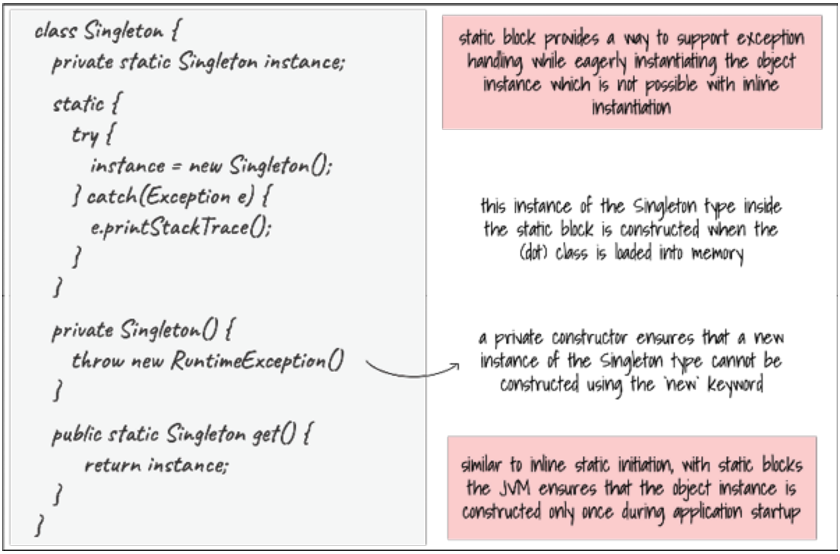
```
class Singleton {
  private static Singleton instance = new Singleton();
  private Singleton() {}
  public static Singleton get() {
    return instance;
  }
}
```

eager initialization
returns the pre constructed instance
the static initializer ensures that the instance is constructed when the class is loaded by JVM

***Figure 2.13**: Eager initialization*

The drawback of eager initialization is that the instance is always constructed irrespective of whether it will be used in runtime or not. If the object construction process throws an exception it can cause the application startup to fail depending on how the construction process has been handled.

Figure 2.14, exhibits another way of using eager initialization, using static blocks:



```
class Singleton {
  private static Singleton instance;

  static {
    try {
      instance = new Singleton();
    } catch(Exception e) {
      e.printStackTrace();
    }
  }

  private Singleton() {
    throw new RuntimeException()
  }

  public static Singleton get() {
      return instance;
  }
}
```

static block provides a way to support exception handling while eagerly instantiating the object instance which is not possible with inline instantiation

this instance of the Singleton type inside the static block is constructed when the (dot) class is loaded into memory

a private constructor ensures that a new instance of the Singleton type cannot be constructed using the 'new' keyword

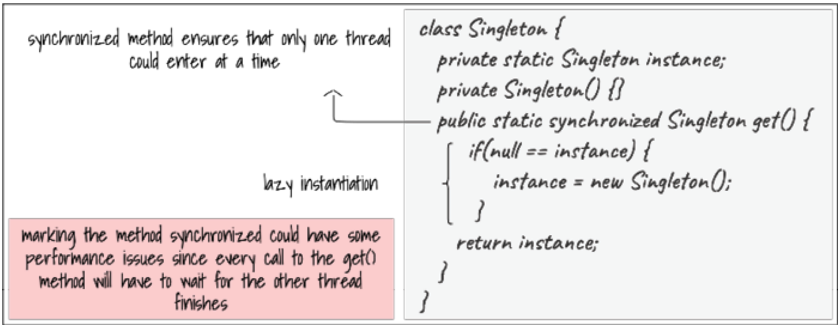
similar to inline static initiation, with static blocks the JVM ensures that the object instance is constructed only once during application startup

Figure 2.14*: Static block eager initialization*

The singleton object that we want to construct could throw an exception during initialization and halt the execution of the application. Though we cannot complete the object construction if an exception is thrown, we can handle the exception in the desired manner. Both these constructs initialize the singleton during class loading by the JVM and safeguard from multiple instance creation.

Thread-safe singleton

A thread-safe singleton shown in *Figure 2.15* makes use of synchronization to allow sequential access to threads, thus ensuring that only one instance of the singleton gets constructed, and all other threads get access to the already initialized instance of the singleton:



synchronized method ensures that only one thread could enter at a time

lazy instantiation

marking the method synchronized could have some performance issues since every call to the get() method will have to wait for the other thread finishes

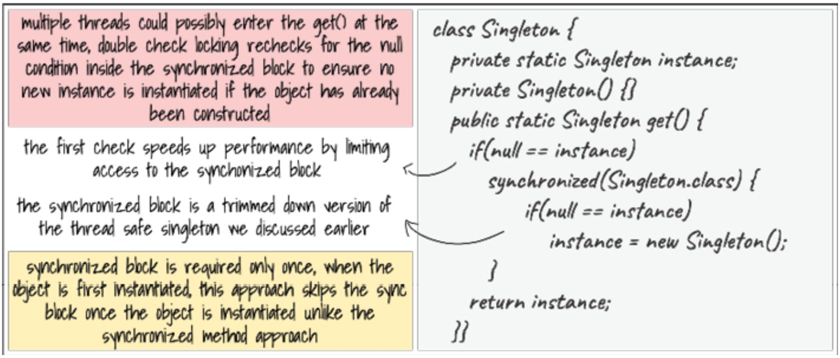
```
class Singleton {
  private static Singleton instance;
  private Singleton() {}
  public static synchronized Singleton get() {
    if(null == instance) {
      instance = new Singleton();
    }
   return instance;
  }
}
```

Figure 2.15: Thread-safe singleton

The thread-safe singleton has a drawback when it comes to performance. Making the accessor method synchronized serves the purpose of ensuring sequential access to threads, but it also restricts access to the method. Due to this limitation, a thread must wait for its chance to enter the method and access the already constructed instance.

The mentioned limitation can be removed by using synchronized blocks instead of a synchronized method. By using the synchronized block, we can limit the amount of code that needs synchronization and reduce the wait time for other threads. When the method is synchronized, threads must wait for their turn to enter the method, but with double-check locking, that is no longer required.

Figure 2.16 exhibits the use of double-check locking to improve the implementation of a thread-safe singleton:



multiple threads could possibly enter the get() at the same time, double check locking rechecks for the null condition inside the synchronized block to ensure no new instance is instantiated if the object has already been constructed

the first check speeds up performance by limiting access to the synchronized block

the synchronized block is a trimmed down version of the thread safe singleton we discussed earlier

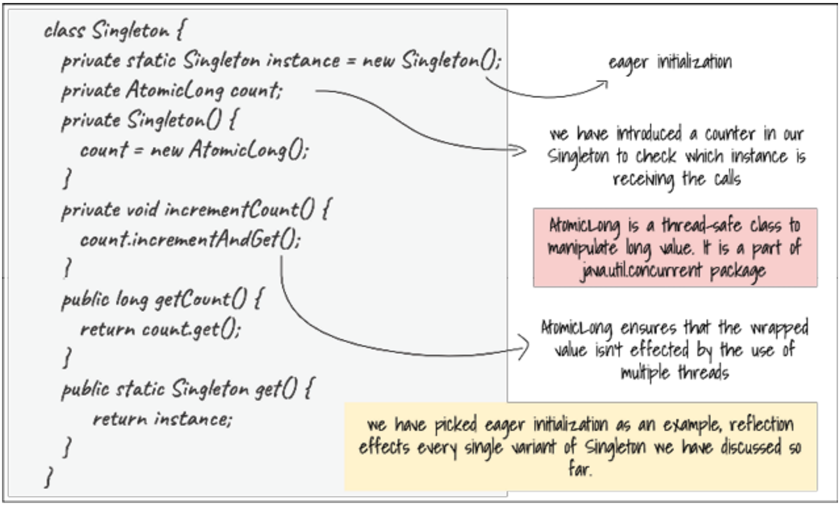
synchronized block is required only once, when the object is first instantiated, this approach skips the sync block once the object is instantiated unlike the synchronized method approach

```
class Singleton {
  private static Singleton instance;
  private Singleton() {}
  public static Singleton get() {
    if(null == instance)
      synchronized(Singleton.class) {
        if(null == instance)
          instance = new Singleton();
      }
    return instance;
  }}
```

Figure 2.16: Thread-safe singleton | Double-check locking

Reflection and singleton

Reflection API in Java allows us to inspect and modify attributes of interfaces, classes, fields, and methods. *Figure 2.17* exhibits a modified version of our singleton class to test the effect of reflection on singleton objects:



```
class Singleton {
  private static Singleton instance = new Singleton();
  private AtomicLong count;
  private Singleton() {
    count = new AtomicLong();
  }
  private void incrementCount() {
    count.incrementAndGet();
  }
  public long getCount() {
    return count.get();
  }
  public static Singleton get() {
      return instance;
  }
}
```

eager initialization

we have introduced a counter in our Singleton to check which instance is receiving the calls

AtomicLong is a thread-safe class to manipulate long value. It is a part of java.util.concurrent package

AtomicLong ensures that the wrapped value isn't effected by the use of multiple threads

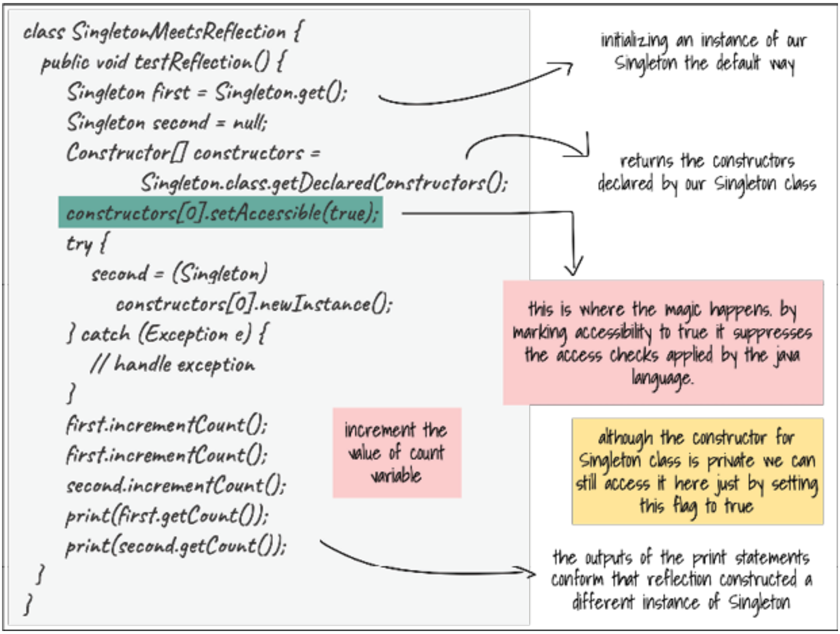
we have picked eager initialization as an example, reflection effects every single variant of Singleton we have discussed so far.

***Figure 2.17**: Reflection and singleton I*

Reflection gives access to class constructors and overloaded methods. We can construct instances of a type, invoke methods, and set field values at runtime. All this, and much more, can be done during runtime.

Reflection can be used to break down the principle of singletons, i.e., to have only one instance of a type. In *Figure 2.16*, we modified our singleton class and added some methods to help us understand if reflection really can create multiple instances of a type even when it is accessible only through a singleton.

Refer to the following example to build a comprehensive understanding. In *Figure 2.18*, we have constructed an instance of our singleton class using reflection:



```
class SingletonMeetsReflection {
  public void testReflection() {
     Singleton first = Singleton.get();
     Singleton second = null;
     Constructor[] constructors =
                 Singleton.class.getDeclaredConstructors();
     constructors[0].setAccessible(true);
     try {
        second = (Singleton)
            constructors[0].newInstance();
     } catch (Exception e) {
        // handle exception
     }
     first.incrementCount();
     first.incrementCount();
     second.incrementCount();
     print(first.getCount());
     print(second.getCount());
  }
}
```

initializing an instance of our Singleton the default way

returns the constructors declared by our Singleton class

this is where the magic happens. by marking accessibility to true it suppresses the access checks applied by the java language.

increment the value of count variable

although the constructor for Singleton class is private we can still access it here just by setting this flag to true

the outputs of the print statements conform that reflection constructed a different instance of Singleton

***Figure 2.18**: Reflection and singleton II*

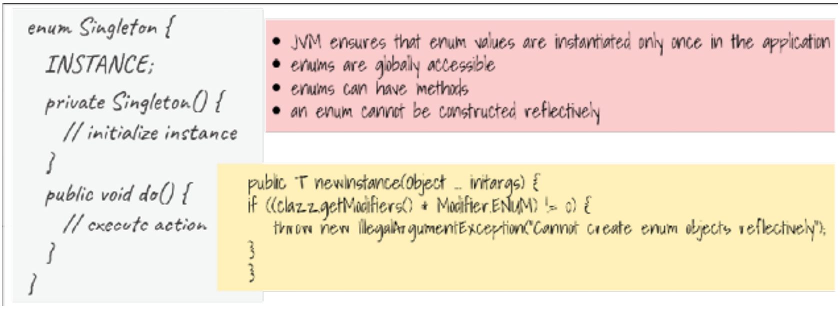
Although the constructor for our class is private, we can still access it by marking it accessible via reflection. This gives us total control over the class and allows us to construct another instance of it, ignoring its singleton property.

In the last lines of the example, we print the value of the count variable to confirm whether reflection breaks apart the singleton design pattern. Those two lines print two and one, respectively, that is, the number of calls we made with the two instances. The *first* instance was created by using the default accessor method, while the *second* one was constructed using reflection. They should have printed three and three if the principle of singleton had held.

Enum as a singleton

While reflection can break apart all the discussed variants of singleton, there is still a way to ensure that the singleton object remains unique. By using an enum, one can ensure that the singleton object is unique in the entire application. In *Figure 2.19*, the code block on the right side is taken from the

java.lang.reflection.Constructor class, which explicitly mentions that an enum cannot be constructed reflectively:



```
enum Singleton {
  INSTANCE;
  private Singleton() {
    // initialize instance
  }
  public void do() {
    // execute action
  }
}
```

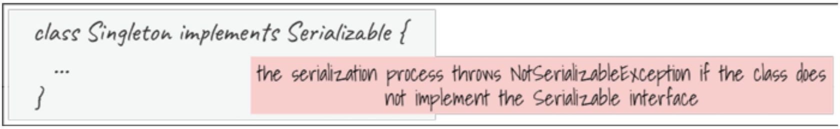
- JVM ensures that enum values are instantiated only once in the application
- enums are globally accessible
- enums can have methods
- an enum cannot be constructed reflectively

```
public T newInstance(Object ... initargs) {
if ((clazz.getModifiers() & Modifier.ENUM) != 0) {
   throw new IllegalArgumentException("Cannot create enum objects reflectively");
}
}
```

Figure 2.19: Enum and singleton

Singleton and serialization

Serialization is the process of converting the state of a Java object into a byte stream that could be stored and later read and reverted into a copy of the serialized object. For a singleton to support serialization, it must implement the serializable interface as depicted in *Figure 2.20*:



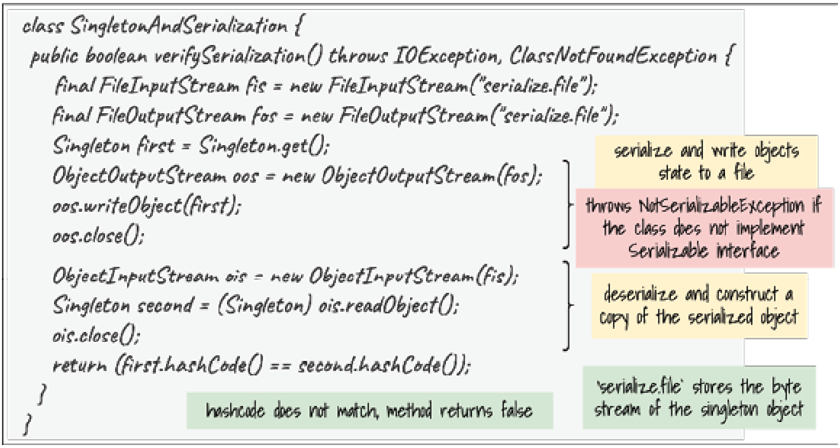
```
class Singleton implements Serializable {
  ...
}
```

the serialization process throws NotSerializableException if the class does not implement the Serializable interface

Figure 2.20: Serialized singleton

The process of converting the serialized form into a copy of the object is called **deserialization**. It constructs a copy of the original object; it is not the same object that we serialized.

In *Figure 2.21*, we first serialize our singleton object, save it, and then read it to construct a copy of the original object that we serialized previously:



```
class SingletonAndSerialization {
 public boolean verifySerialization() throws IOException, ClassNotFoundException {
   final FileInputStream fis = new FileInputStream("serialize.file");
   final FileOutputStream fos = new FileOutputStream("serialize.file");
   Singleton first = Singleton.get();
   ObjectOutputStream oos = new ObjectOutputStream(fos);
   oos.writeObject(first);
   oos.close();

   ObjectInputStream ois = new ObjectInputStream(fis);
   Singleton second = (Singleton) ois.readObject();
   ois.close();
   return (first.hashCode() == second.hashCode());
 }
}
```

serialize and write objects state to a file

throws NotSerializableException if the class does not implement Serializable interface

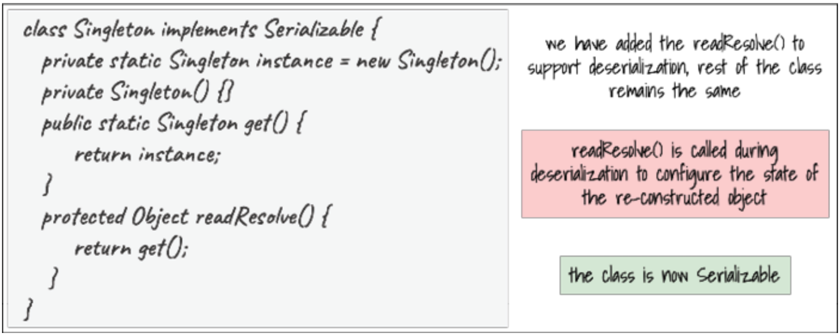
deserialize and construct a copy of the serialized object

'serialize.file' stores the byte stream of the singleton object

hashcode does not match, method returns false

***Figure 2.21**: Singleton and serialization*

Why does the deserialized object not match the original one? Is not implementing serializable the only thing to do? To preserve the state of the object, we must make some additional changes to the class, as illustrated in *Figure 2.22*:



```
class Singleton implements Serializable {
  private static Singleton instance = new Singleton();
  private Singleton() {}
  public static Singleton get() {
      return instance;
  }
  protected Object readResolve() {
      return get();
  }
}
```

we have added the readResolve() to support deserialization, rest of the class remains the same

readResolve() is called during deserialization to configure the state of the re-constructed object

the class is now Serializable

***Figure 2.22**: Read resolve | Singleton and reflection*

The updated **Singleton** class constructs an equivalent object during the deserialization process. After making this change, the hash code of the original and the reconstructed object should match. By adding the implementation of the **readResolve** method, in this manner, we can preserve the state of our singleton and ensure that it does not break during serialization.

Enums are inherently serializable, and there is no need to add **readResolve** to them. All the enums by default implement the serializable interface and can be serialized and deserialized in the same way.

Advantages and drawbacks

The singleton design pattern has many advantages in comparison to the objects that are created on-demand. Let us discuss the benefits first and then focus on the drawbacks:

- It promotes reusability.
- It helps to efficiently manage resource consumption by allowing the creation of only a single object of a type.
- It provides a single interaction point and allows direct access to it.

The drawbacks of the singleton design pattern are as follows:

- Interacting directly with the singleton results in tight coupling, which can be difficult to remove later.
- Singletons can be difficult to mock when testing a class that accesses them.
- A singleton is like a global object; therefore, any change in its state can impact all the other objects or threads accessing it.

Usage

The following list outlines some of the scenarios relevant to singleton design patterns:

- It can be used for creating objects that consume a lot of resources or time when constructed.
- It is best suited for objects that provide services to other objects, and their state is seldom changed.

Conclusion

The singleton design pattern is one of the most used design patterns and is also easy to understand and program. It is widely used in scenarios that require a single instance throughout the application.

There can only be one unique instance of that type. It should be accessible from anywhere within the application, and it should support concurrent access to the resources it safeguards. In this chapter, we discussed the basics of singleton design patterns, the different ways they can be constructed, and how they could be easily broken apart. We also discussed the remedies to safeguard the singleton from such breaks.

In the next chapter, we will discuss another creational design pattern, the factory design pattern, and how it can be utilized to create abstract objects, allowing for loose coupling and the need for the client to know the specific implementation classes.

Questions

In this section, we will sharpen our brains by indulging in some fun brain activities. Let us get started.

Find the odd one out

1. Lazy initialization, thread-safe singleton, eager initialization, double check locking
2. Eager initialization, enum as a singleton, lazy initialization, thread-safe singleton
3. Serializable, readResolve, Enum, NotSerializableException
4. Enum as a singleton:
    a. Is thread-safe
    a. Allows lazy initialization
    b. Can be constructed using reflection
    c. Do not support serialization
5. Singletons:
    a. Must support multi-thread applications
    b. Are globally accessible
    c. Can have multiple instances in an application
    d. Are one-of-a-kind objects

Select two correct answers

1. Eager initialization:
    a. Is thread-safe
    b. Cannot be broken via reflection
    c. This can be done via static blocks
    d. Supports lazy loading
2. Reflection can be used to:
    a. Check object inheritance
    b. Invoke methods
    c. Construct the instance of an enum
    d. Change field types
3. Serialization:
    a. Is a method to convert objects into a byte stream
    b. Is a method to convert a byte stream into an object
    c. Is supported by an enum
    d. This can be done without implementing serializable
4. What does together:
    a. Enum, serialization, eager initialization
    b. Thread-safe, enum, lazy loading
    c. Support concurrency, unique, globally accessible
    d. Lazy initialization, static blocks, enum

Answers

This section provides the answers to the questions given.

Find the odd one out

1. Thread-safe singleton
2. Enum as a singleton
3. Enum
4. Is thread-safe

5. Can have multiple instances in an application

Select two correct answers

1. Is thread-safe, can be done via static blocks
2. Invoke methods, construct an instance of an enum
3. Is a method to convert an object into a byte stream, which is supported by enum
4. Enum, serialization, eager initialization and, support concurrency, unique, globally accessible

Join our Discord space

Join our Discord workspace for latest updates, offers, tech happenings around the world, new releases, and sessions with the authors:

https://discord.bpbonline.com

CHAPTER 3

Object Factory

Introduction

Abstraction plays an important role in the way objects interact. It provides the feasibility of introducing changes to the application without affecting the existing structure in a significant manner. Abstraction is a key component for a loosely coupled, low-maintenance, and adaptive architecture.

Abstraction can be achieved by using abstract classes or interfaces. Coding to interfaces or abstract classes reduces coupling among classes, allows for dynamic polymorphism, and enables runtime injection of concrete instances to their parent type references. With abstraction in place, new implementations can be easily introduced into the existing system with minimal changes.

In this chapter, we will go through examples that demonstrate the factory design pattern and the role of abstraction and polymorphism in it.

Structure

This chapter covers the following topics:

- Interacting with interfaces
- Building factories
- Response factory
- Multiple factories
- Factory method pattern
- Family of objects
- Abstract factory
- Advantages and drawbacks
- Usage

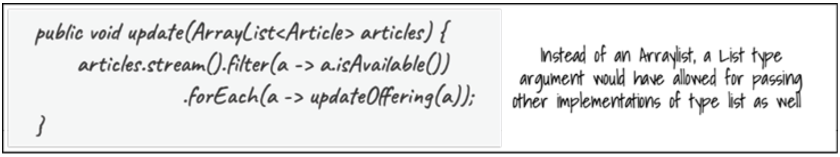
Objectives

By the end of this chapter, the readers will be able to understand the use of abstraction and polymorphism to reduce coupling. The readers will learn about the factory design patterns and the application of polymorphism to bring them to life. They will also learn about the simple factory, the factory method, and the abstract factory pattern.

Interacting with interfaces

Objects are the basic building blocks in an object-oriented programming language. They store data, interact with each other, and process information. Interaction between objects is an essential part of the application. It is hard to think of an application that has a single object type with no interaction requirements. The way the objects interact becomes more important when the applications grow and introduce new types of objects. These new object types, when incorporated into the existing codebase, require changes that there might be, few or many, depending on the way the objects interact. If the objects are tightly coupled to each other, they are hard to separate and often require a great deal of effort.

The way the objects are coupled together defines the capability of the application to adapt to changes. Tight coupling signifies interaction among or with concrete implementations. As mentioned earlier, tight coupling should be avoided to minimize the effort required to introduce changes into the application. *Figure 3.1* exhibits one such use case where a concrete implementation of type **List** is passed to a method:



```
public void update(ArrayList<Article> articles) {
    articles.stream().filter(a -> a.isAvailable())
                     .forEach(a -> updateOffering(a));
}
```

Instead of an Arraylist, a List type argument would have allowed for passing other implementations of type list as well

***Figure 3.1**: Interacting with interfaces*

Coding to implementation binds classes together and restricts the ability for dynamic coupling. This also means that the code is never closed for modification and will have to be reopened to incorporate changes.

Coding to interfaces keeps the classes loosely coupled and unaware of their connection with other concrete classes until runtime, when polymorphism comes into action, and the interface reference is substituted with a concrete subtype. Interfaces provide the desired abstraction, and polymorphism allows concrete implementations to substitute an interface-type reference.

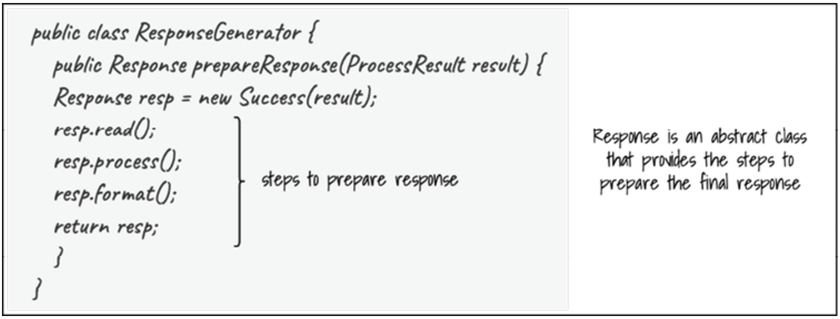
We cannot eliminate the need for coding implementations. Eventually, there will be direct interactions with the concrete types. Such interactions can be reduced by coding to interfaces and utilizing polymorphism, leaving less code open for modification. The concrete implementations that implement the desired interfaces can be easily incorporated into the application without much change.

Building factories

This section discusses an example of a *report generation system* that prepares reports for a wide variety of use cases; reports that are generated after processing input data, in near real-time, and that can be downloaded via a browser. With the increase in traffic and the amount of data that should be processed, the system is under stress, and the users are experiencing longer processing times and abrupt, erroneous behavior.

As a solution to this problem, the system can be redesigned to introduce scheduled processing of input data and to allow the users to visit the application, schedule the reports, and get notified once the report is ready. For the needs of this chapter, we will only focus on the report generation aspect of the overall change description.

Let us start with a simple response generator and then systematically improve it so that it can be generic enough to produce responses for multiple use cases. *Figure 3.2* exhibits the **ResponseGenerator** class that prepares one of the many response types, i.e., **Success. Response** is an abstract class that provides steps to prepare a response in the form of methods. It is also the parent class of the concrete type, **Success**.



```
public class ResponseGenerator {
   public Response prepareResponse(ProcessResult result) {
   Response resp = new Success(result);
   resp.read();
   resp.process();
   resp.format();
   return resp;
   }
}
```

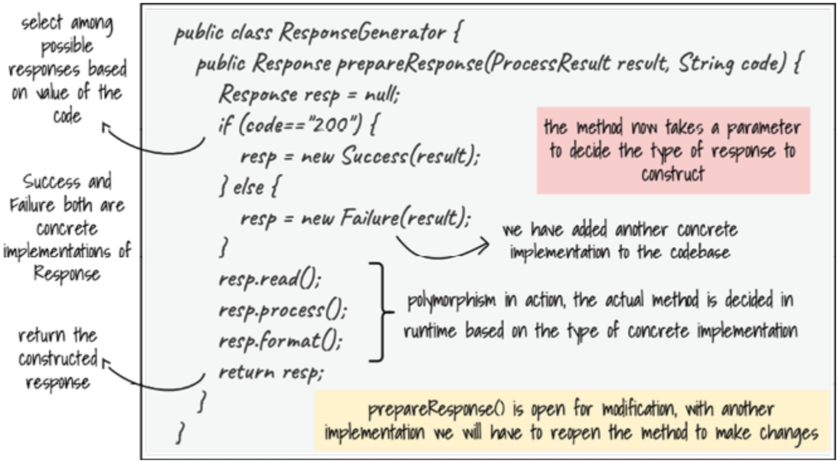
steps to prepare response

Response is an abstract class that provides the steps to prepare the final response

***Figure 3.2**: Response generator I*

If you are wondering why we need this class when only a single type of response is being generated, imagine other types of responses being added to the application at a later stage. Creating methods for every single response type does not seem to be a good idea, instead, if we could have a parent type that is extended by all the different types of response, then returning a polymorphic reference from the method will be sufficient for all response types. **Response** class provides us with the needed abstraction and allows for polymorphic substitution.

A response generator that always returns success looks rather lame, let us add some more response types to make it look smart and close to reality. For starters, let us add a response type for failure as well. In *Figure 3.3*, the method now returns two different types of response, success, and failure:



```
public class ResponseGenerator {
   public Response prepareResponse(ProcessResult result, String code) {
      Response resp = null;
      if (code=="200") {
         resp = new Success(result);
      } else {
         resp = new Failure(result);
      }
      resp.read();
      resp.process();
      resp.format();
      return resp;
   }
}
```

select among possible responses based on value of the code

Success and Failure both are concrete implementations of Response

return the constructed response

the method now takes a parameter to decide the type of response to construct

we have added another concrete implementation to the codebase

polymorphism in action, the actual method is decided in runtime based on the type of concrete implementation

prepareResponse() is open for modification, with another implementation we will have to reopen the method to make changes

***Figure 3.3**: Response generator II*

The method is modified to take a string as an input parameter to decide the type of response. The if else directive selects the correct response type to be constructed, and then finally, the response is prepared.

Every time there is a need to add another type of response, the response generation class should be revisited to introduce the changes. In short, the class is open for modification. A class that is open for modification is vulnerable to bugs and needs more effort to test. To ensure correct and expected behavior, the whole class must be tested and not just the changes. As a guiding principle, a class should be open for extension but closed for modification.

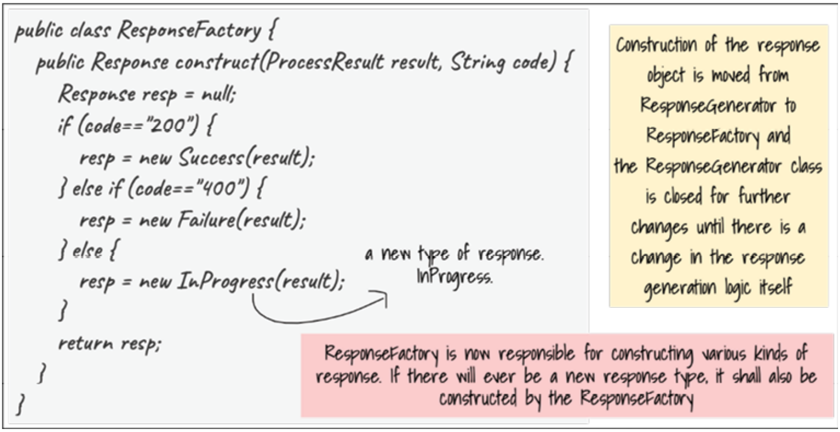
While we mentioned the *open-close principle*, let us answer some questions in relation to its usage in the classes we mentioned earlier:

1. *What if we shift the code that is vulnerable to change to some other class?* By moving the vulnerable code to another class, we can restrict the changes to the target class and close the current class for changes.
2. *Will the* **ResponseGenerator** *class be truly close to changes? What if a new step is added to prepare a response?* The answer is no, because if a new step is added to prepare a response, then our **ResponseGenerator** class will have to be modified, as it is still responsible for the response generation logic.

3. *Can the* **Failure** *response have preparation steps different from other response types? Can we call those steps from our* **ResponseGenerator** *class?* Since the reference type in the **ResponseGenerator** class is **Response**, only those methods exposed by it are visible to the **ResponseGenerator**. **Failure** response can have methods for steps specific to its generation that can be called from within the overridden ones to ensure those steps are processed. If a step is common in all the concrete types, then only it should be added to the **Response** class; otherwise, not.

Response factory

Let us follow the principle discussed earlier, close the response generator class for modification, and move the logic to construct different types of responses to another class. This new class will be responsible for constructing objects of various response types. *Figure 3.4* exhibits our first impression of a factory:



```
public class ResponseFactory {
  public Response construct(ProcessResult result, String code) {
    Response resp = null;
    if (code=="200") {
      resp = new Success(result);
    } else if (code=="400") {
      resp = new Failure(result);
    } else {
      resp = new InProgress(result);
    }
    return resp;
  }
}
```

a new type of response. InProgress.

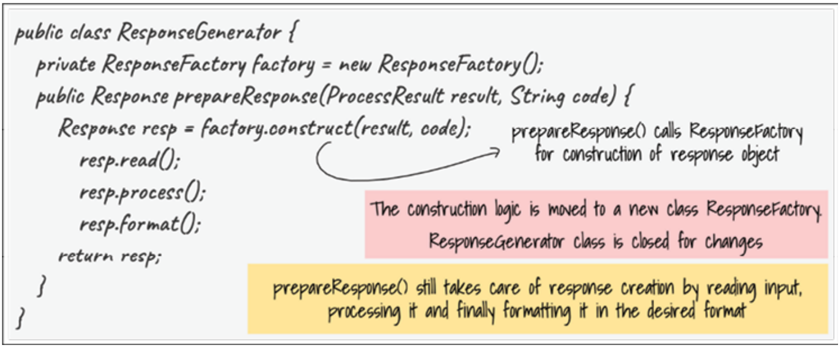
Construction of the response object is moved from ResponseGenerator to ResponseFactory and the ResponseGenerator class is closed for further changes until there is a change in the response generation logic itself

ResponseFactory is now responsible for constructing various kinds of response. If there will ever be a new response type, it shall also be constructed by the ResponseFactory

***Figure 3.4**: Response factory I*

A factory is essentially responsible for creating objects of a particular type, and in doing so, it can make use of other interfaces or classes down the line. A factory provides the necessary abstraction by hiding the creation logic from its user. It also provides centralized control over the object creation process that can be utilized for data validation, permission verification, or something else entirely. For example, a factory could be a place to decide whether the user has all the necessary permissions to access and use an object.

The response factory class is a basic implementation of a factory that can construct objects of type response. If there is a need for a new type of response, it should be added to the factory as well. Let us now modify our response generator class and close it for further changes. *Figure 3.5* exhibits the changes made to the response generator class; it now has a reference to the response factory that is responsible for constructing different types of responses:



```
public class ResponseGenerator {
  private ResponseFactory factory = new ResponseFactory();
  public Response prepareResponse(ProcessResult result, String code) {
    Response resp = factory.construct(result, code);
      resp.read();
      resp.process();
      resp.format();
    return resp;
  }
}
```

prepareResponse() calls ResponseFactory for construction of response object

The construction logic is moved to a new class ResponseFactory.
ResponseGenerator class is closed for changes

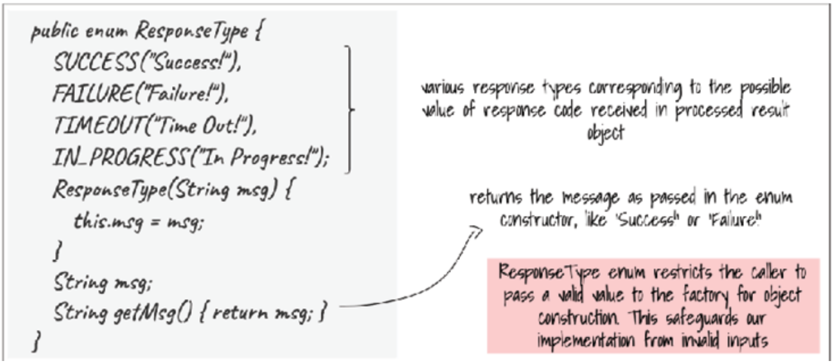
prepareResponse() still takes care of response creation by reading input, processing it and finally formatting it in the desired format

***Figure 3.5**: Response generator and factory*

The **ResponseFactory** class exposes a method called **construct**, which is called by the response generator for object creation. The selection of the type of response is done by passing a parameter to the factory, as was done in the generator.

In our newly constructed factory class, the decision parameter is of type string that could be passed an invalid value, one that is not expected, failing the construction process. To resolve this, we can restrict the values that can be passed to the **construct** method by creating an enum that contains all the possible values relative to the response types.

As exhibited in *Figure 3.6*, the enum can be used to restrict the input values that are passed to the **construct** method:



```
public enum ResponseType {
    SUCCESS("Success!"),
    FAILURE("Failure!"),
    TIMEOUT("Time Out!"),
    IN_PROGRESS("In Progress!");
    ResponseType(String msg) {
        this.msg = msg;
    }
    String msg;
    String getMsg() { return msg; }
}
```

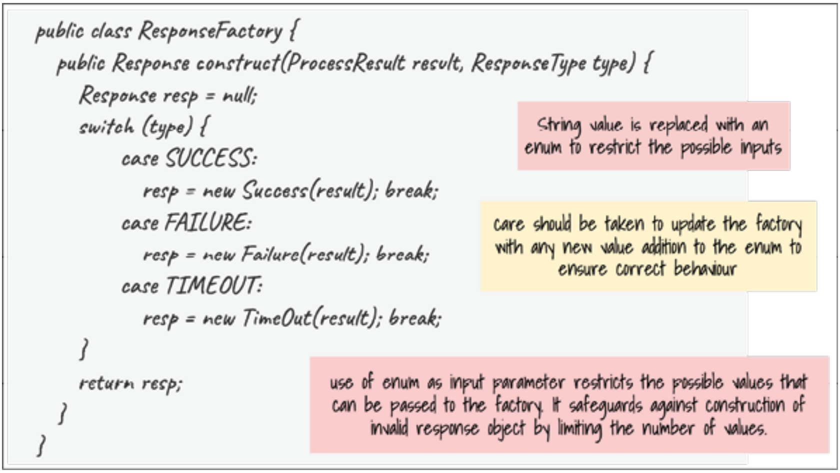
various response types corresponding to the possible value of response code received in processed result object

returns the message as passed in the enum constructor, like 'Success!' or 'Failure!'

ResponseType enum restricts the caller to pass a valid value to the factory for object construction. This safeguards our implementation from invalid inputs

Figure 3.6: *Response type enum*

In order to ensure that our factory is aware of any new response type and behaves correctly, it should be kept in sync with the enum at all times. *Figure 3.7* exhibits the changes required in the **ResponseFactory** class:



```
public class ResponseFactory {
    public Response construct(ProcessResult result, ResponseType type) {
        Response resp = null;
        switch (type) {
            case SUCCESS:
                resp = new Success(result); break;
            case FAILURE:
                resp = new Failure(result); break;
            case TIMEOUT:
                resp = new TimeOut(result); break;
        }
        return resp;
    }
}
```

String value is replaced with an enum to restrict the possible inputs

care should be taken to update the factory with any new value addition to the enum to ensure correct behaviour

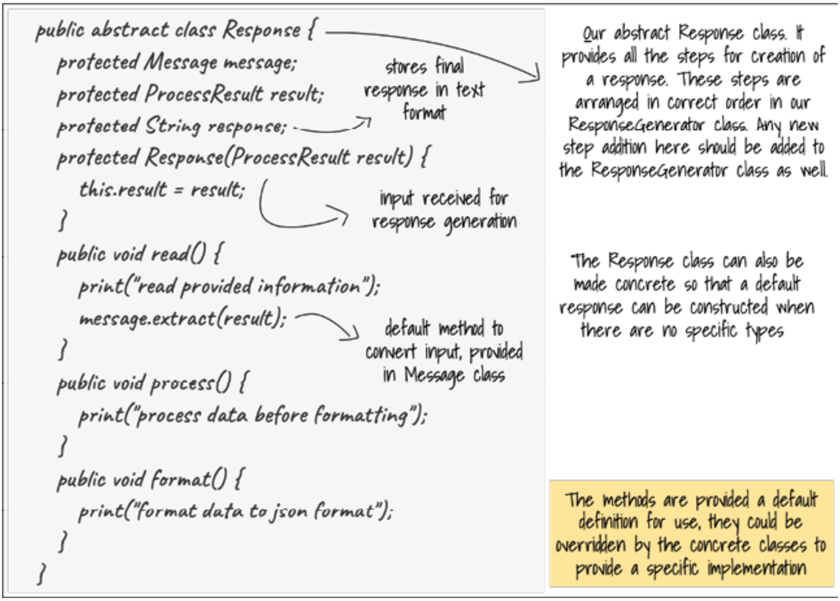
use of enum as input parameter restricts the possible values that can be passed to the factory. It safeguards against construction of invalid response object by limiting the number of values.

Figure 3.7: *Response factory II*

One fact that we have not yet discussed is the abstract class response and the concrete classes that implement it. Let us get familiar with those classes before we move ahead; they are important to understand the rest of this chapter.

Figure 3.8 exhibits the abstract **Response** class. It provides the steps in the form of methods in order to prepare a response. The **Message** class stores the information after parsing the **input**, and the

response is the final processed response string.



```
public abstract class Response {
   protected Message message;
   protected ProcessResult result;
   protected String responce;
   protected Response(ProcessResult result) {
      this.result = result;
   }
   public void read() {
      print("read provided information");
      message.extract(result);
   }
   public void process() {
      print("process data before formatting");
   }
   public void format() {
      print("format data to json format");
   }
}
```

stores final response in text format

input received for response generation

default method to convert input, provided in Message class

Our abstract Response class. It provides all the steps for creation of a response. These steps are arranged in correct order in our ResponseGenerator class. Any new step addition here should be added to the ResponseGenerator class as well.

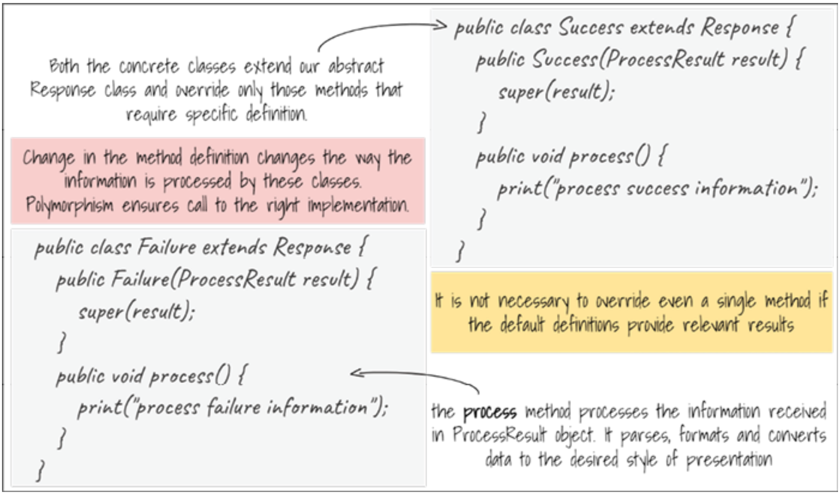
The Response class can also be made concrete so that a default response can be constructed when there are no specific types

The methods are provided a default definition for use, they could be overridden by the concrete classes to provide a specific implementation

***Figure 3.8**: Abstract response class*

Since the class is abstract, we cannot construct its object directly. If a generic response is required, a concrete class can be used instead. Notice that the class has a default definition for all the methods; these generic definitions can be utilized by the concrete implementations, or they can be overridden. To ensure that the implementations provide their own definition of these methods, they should be made abstract.

The order in which these steps should be called is defined in the **ResponseGenerator** class, which is responsible for preparing the final response. The response factory is utilized to construct the response. The response generator class is the only place where this arrangement should be maintained to ensure that each response is prepared following the same steps. *Figure 3.9* demonstrates the concrete classes that form the actual response:



Both the concrete classes extend our abstract Response class and override only those methods that require specific definition

```
public class Success extends Response {
    public Success(ProcessResult result) {
        super(result);
    }
    public void process() {
        print("process success information");
    }
}
```

Change in the method definition changes the way the information is processed by these classes. Polymorphism ensures call to the right implementation.

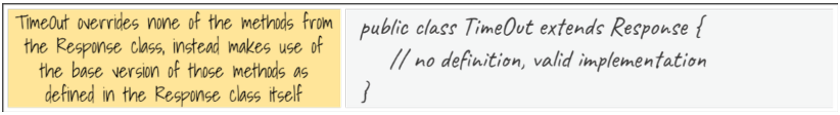
```
public class Failure extends Response {
    public Failure(ProcessResult result) {
        super(result);
    }
    public void process() {
        print("process failure information");
    }
}
```

It is not necessary to override even a single method if the default definitions provide relevant results

the **process** method processes the information received in ProcessResult object. It parses, formats and converts data to the desired style of presentation

***Figure 3.9**: Concrete response class: Success*

Notice that not all methods are overridden; instead, only those that require a different definition are rewritten in the concrete implementations. Since we have already provided default implementations for all the methods in our response class, these definitions are used when a method is not. In *Figure 3.10*, even though the class does not override any of the parent methods, it is still a valid implementation:



TimeOut overrides none of the methods from the Response class, instead makes use of the base version of those methods as defined in the Response class itself

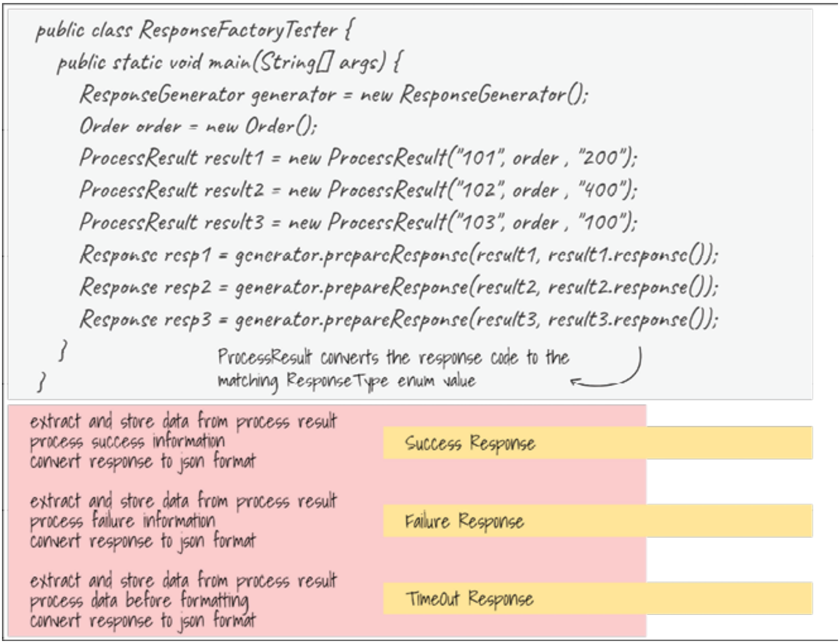
```
public class TimeOut extends Response {
    // no definition, valid implementation
}
```

***Figure 3.10**: Concrete response class: TimeOut*

First run of response factory

Our test class demonstrates the use of a response factory that works behind the scenes to create objects of different concrete types based on the enum value passed to the factory. It also demonstrates the use of polymorphism for selecting the correct implementation.

In *Figure 3.11*, the test class instantiates a response generator and calls one of its methods, passing in the desired input and the type of response it should prepare. It is worth noting that we have passed the enum value explicitly to the method, while it is a part of the input itself; this is deliberately done for clarity and readability.



```
public class ResponseFactoryTester {
  public static void main(String[] args) {
    ResponseGenerator generator = new ResponseGenerator();
    Order order = new Order();
    ProcessResult result1 = new ProcessResult("101", order , "200");
    ProcessResult result2 = new ProcessResult("102", order , "400");
    ProcessResult result3 = new ProcessResult("103", order , "100");
    Response resp1 = generator.prepareResponse(result1, result1.response());
    Response resp2 = generator.prepareResponse(result2, result2.response());
    Response resp3 = generator.prepareResponse(result3, result3.response());
  }
}
```

ProcessResult converts the response code to the matching ResponseType enum value
extract and store data from process result
process success information
convert response to json format
Success Response
extract and store data from process result
process failure information
convert response to json format
Failure Response
extract and store data from process result
process data before formatting
convert response to json format
TimeOut Response

***Figure 3.11**: Response factory tester I*

The simple factory

The **ResponseFactory** class that we discussed in the preceding section is an example of a **simple factory**, as shown in *Figure 3.12*. It is not considered a design pattern but is widely used in applications.

Due to its simplistic approach and easy implementation, it is often used in scenarios that require a factory-like implementation but with less complexity. Not only that, but a lot of developers also often confuse it with the factory design pattern because of its high resemblance to the actual pattern.

Simple factory and factory design patterns have some similarities; for example, both make use of abstraction to construct concrete objects utilizing a similar framework. One of the differences is that the factory design pattern delegates the responsibility of object creation to the concrete types, but the same is not done by the sample factory pattern.

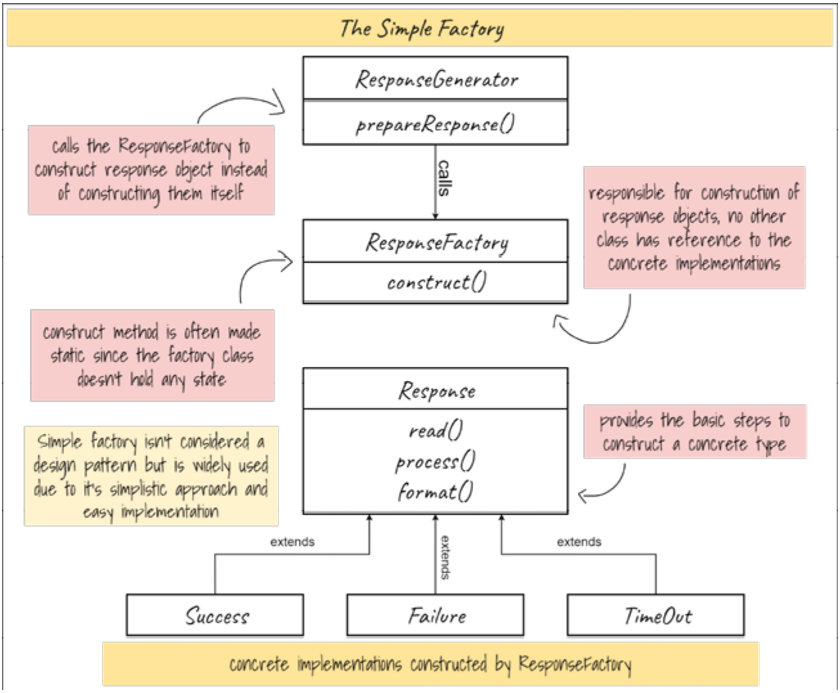

The Simple Factory
ResponseGenerator
prepareResponse()
calls the ResponseFactory to construct response object instead of constructing them itself
calls
ResponseFactory
construct()
responsible for construction of response objects, no other class has reference to the concrete implementations
construct method is often made static since the factory class doesn't hold any state
Response
read()
process()
format()
provides the basic steps to construct a concrete type
Simple factory isn't considered a design pattern but is widely used due to it's simplistic approach and easy implementation
extends
extends
extends
Success
Failure
TimeOut
concrete implementations constructed by ResponseFactory

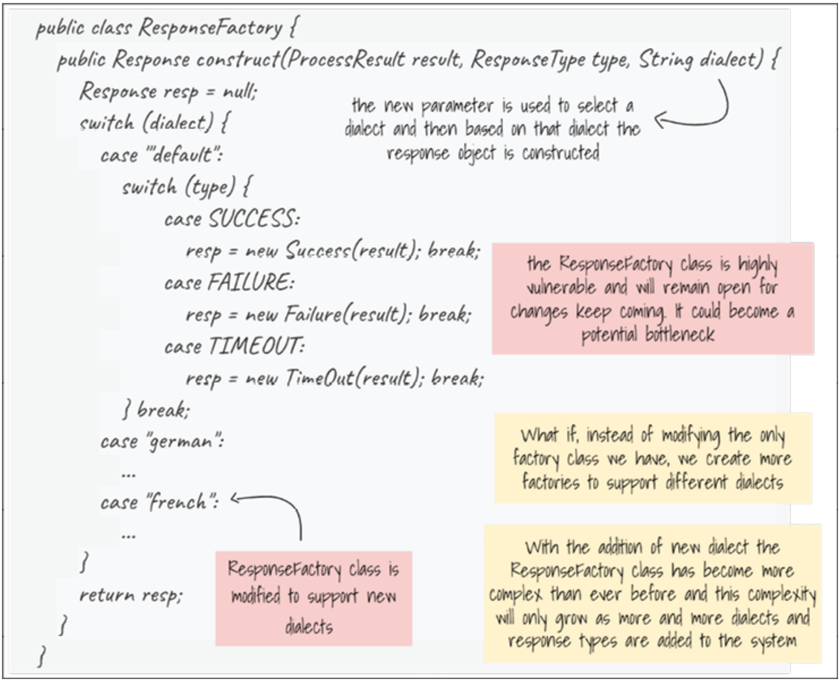
Figure 3.12*: The simple factory*

The simple factory is more like a basic version of the factory design pattern, but an important one. It teaches us the fundamentals of the factory design pattern and prepares us for the underlying complexity of the actual pattern. It should be noted that not all scenarios and use cases exhibit profound complexity and might not require implementing the factory design pattern; for such cases, the simple factory should be considered instead.

The next section focuses on the factory design patterns. We will discuss two of its variations, the factory method pattern and the abstract factory pattern. Both follow similar principles and make intelligent use of abstraction to achieve a loosely coupled design.

Multiple factories

The response factory does a good job dealing with a small number of response types. To take this to the next level, let us assume that the response factory should be made to respond in various languages and not just English, as exhibited in *Figure 3.13*. This change will greatly increase the complexity of the current factory implementation, making it difficult to manage it in the long run.



```
public class ResponseFactory {
   public Response construct(ProcessResult result, ResponseType type, String dialect) {
      Response resp = null;
      switch (dialect) {
        case '"default":
          switch (type) {
              case SUCCESS:
                 resp = new Success(result); break;
              case FAILURE:
                 resp = new Failure(result); break;
              case TIMEOUT:
                 resp = new TimeOut(result); break;
          } break;
        case "german":
          ...
        case "french":
          ...
      }
      return resp;
   }
}
```

the new parameter is used to select a dialect and then based on that dialect the response object is constructed

the ResponseFactory class is highly vulnerable and will remain open for changes keep coming. It could become a potential bottleneck

What if, instead of modifying the only factory class we have, we create more factories to support different dialects

ResponseFactory class is modified to support new dialects

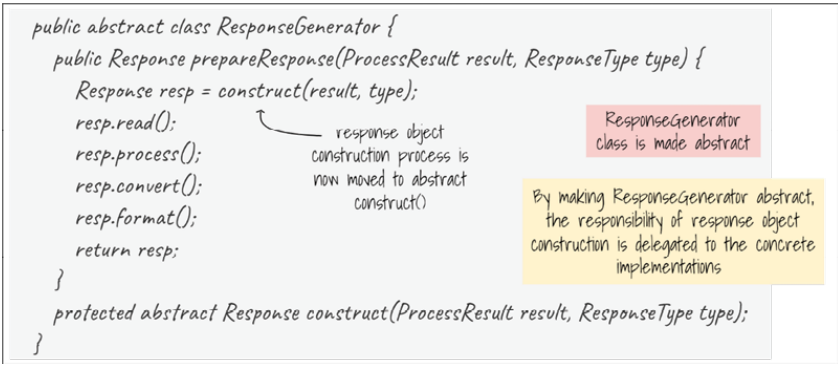
With the addition of new dialect the ResponseFactory class has become more complex than ever before and this complexity will only grow as more and more dialects and response types are added to the system

Figure 3.13: *Response factory | Factory method pattern*

The current implementation requires us to make additions to it every time a new language is introduced. This is a serious drawback that makes it prone to bugs, as the factory implementation is always open to changes. It can be improved by creating multiple factory classes that construct objects of the types of response specific to a language.

The **ResponseGenerator** class, from *Figure 3.5*, can be made abstract, and a new method can be introduced to delegate the object construction process. The factory classes can be modified to implement this class and provide a definition for the new abstract method. The **ResponseGenerator** class will be responsible for preparing the response, but the responsibility of object creation will be transferred to one of its concrete implementations. This also means that the old response factory implementation would no longer be used.

With this change, multiple factory classes can be created and plugged into the application design without modifying the existing logic, all possible because of the loosely coupled design and the use of abstraction and polymorphism. As demonstrated in *Figure 3.14*, the response generator class is not dependent on any of the factories anymore:



```
public abstract class ResponseGenerator {
   public Response prepareResponse(ProcessResult result, ResponseType type) {
      Response resp = construct(result, type);
      resp.read();
      resp.process();
      resp.convert();
      resp.format();
      return resp;
   }
   protected abstract Response construct(ProcessResult result, ResponseType type);
}
```

response object construction process is now moved to abstract construct()

ResponseGenerator class is made abstract

By making ResponseGenerator abstract, the responsibility of response object construction is delegated to the concrete implementations

***Figure 3.14**: Response generator | Factory method pattern*

The concrete factories now extend this class and use it as a base factory. This change has allowed for the development of multiple factory classes. *Figure 3.15* exhibits the application of polymorphism with respect to the response generator implementations:

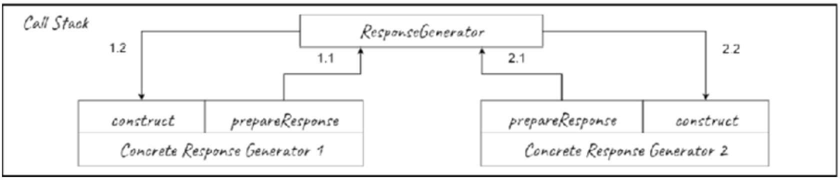
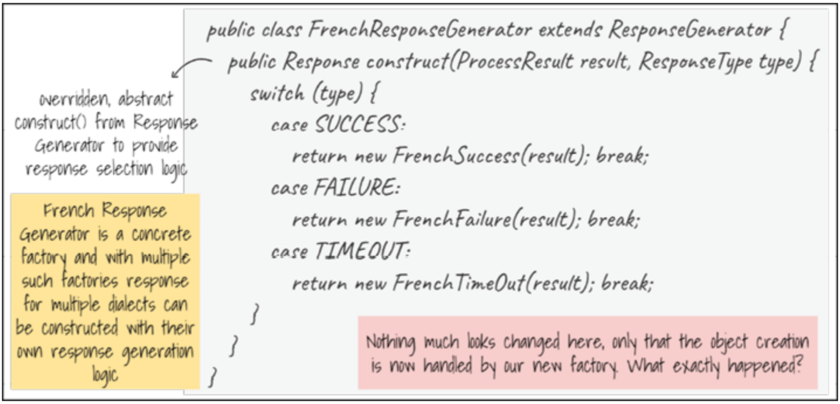


***Figure 3.15**: Polymorphism | Factory method pattern*

Factory that speaks French

Let us create new factory classes that implement the abstract response generator class and add some response types. *Figure 3.16* exhibits one such factory that constructs objects that prepare a response in the French dialect:



```
public class FrenchResponseGenerator extends ResponseGenerator {
  public Response construct(ProcessResult result, ResponseType type) {
    switch (type) {
      case SUCCESS:
        return new FrenchSuccess(result); break;
      case FAILURE:
        return new FrenchFailure(result); break;
      case TIMEOUT:
        return new FrenchTimeOut(result); break;
    }
  }
}
```

overridden, abstract construct() from Response Generator to provide response selection logic

French Response Generator is a concrete factory and with multiple such factories response for multiple dialects can be constructed with their own response generation logic

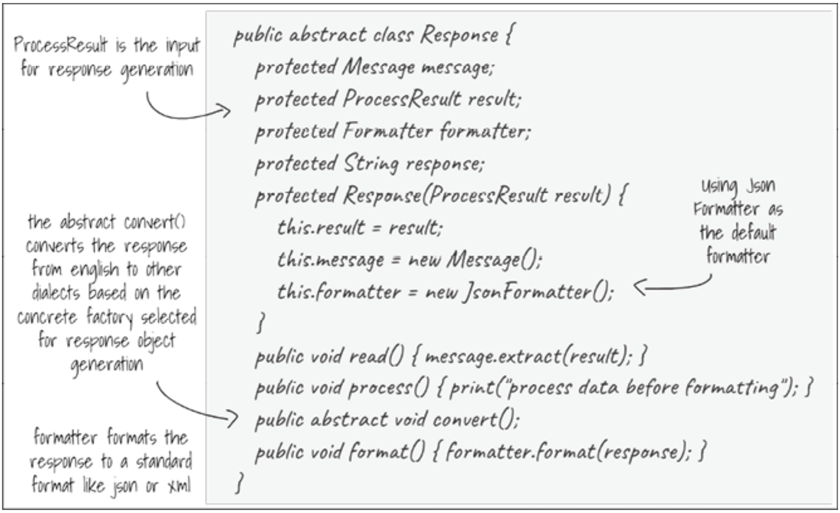
Nothing much looks changed here, only that the object creation is now handled by our new factory. What exactly happened?

***Figure 3.16**: French response generator | Factory method pattern*

This new factory has many similarities with the response factory we created in the previous section. It is essentially the same logic being put in a different class. Notice how the enum is reused without any change; if the existing response factory class is used, multiple new values would have to be added to the enum as well to ensure that every type of response, from every dialect, has a value associated with it.

Response updated

The abstract response class is also updated as shown in *Figure 3.17* to incorporate the changes required to prepare a language-specific response. It now has a new abstract method called **convert**, which converts the response from English, the default language, to the required one.



ProcessResult is the input for response generation

the abstract convert() converts the response from english to other dialects based on the concrete factory selected for response object generation

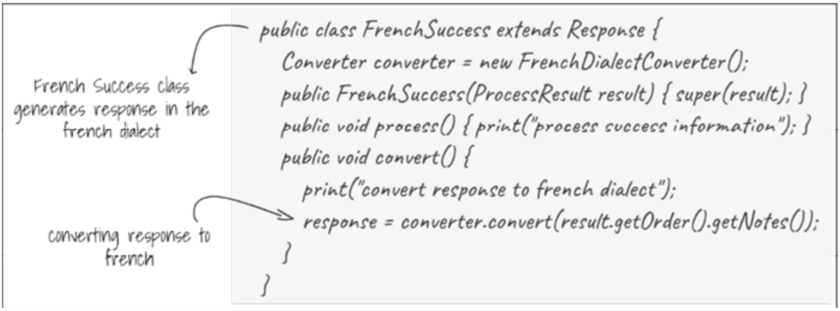
formatter formats the response to a standard format like json or xml

using Json Formatter as the default formatter

```
public abstract class Response {
  protected Message message;
  protected ProcessResult result;
  protected Formatter formatter;
  protected String response;
  protected Response(ProcessResult result) {
    this.result = result;
    this.message = new Message();
    this.formatter = new JsonFormatter();
  }
  public void read() { message.extract(result); }
  public void process() { print(“process data before formatting”); }
  public abstract void convert();
  public void format() { formatter.format(response); }
}
```

Figure 3.17: *Abstract response | Factory method pattern*

Figure 3.18 exhibits a concrete response type that extends the abstract response class and has a reference to a converter implementation to convert from English to French:



French Success class generates response in the french dialect

converting response to french

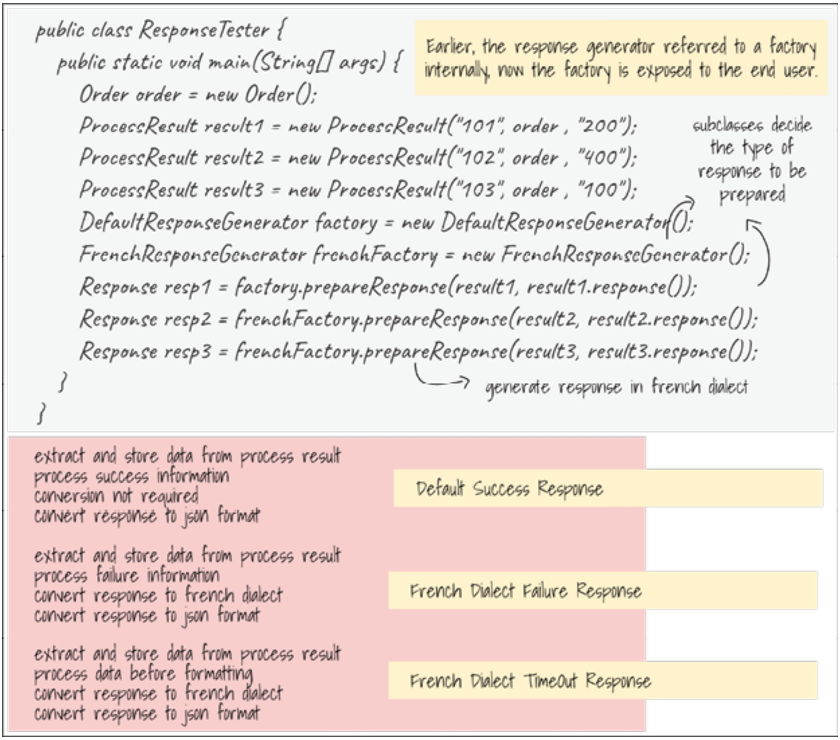
```
public class FrenchSuccess extends Response {
  Converter converter = new FrenchDialectConverter();
  public FrenchSuccess(ProcessResult result) { super(result); }
  public void process() { print(“process success information”); }
  public void convert() {
    print(“convert response to french dialect”);
    response = converter.convert(result.getOrder().getNotes());
  }
}
```

Figure 3.18: *Concrete response French success | Factory method pattern*

There could be as many factories as required, one for each dialect that supports all the possible responses the system could generate, and a different class for each dialect to support it. The updated design allows for better management and easy maintenance of all these concrete implementations.

Second run of response factory

Let us do a test run to verify the changes we have made so far and to finally understand the factory method pattern as shown in *Figure 3.19*:



```
public class ResponseTester {
   public static void main(String[] args) {
      Order order = new Order();
      ProcessResult result1 = new ProcessResult("101", order , "200");
      ProcessResult result2 = new ProcessResult("102", order , "400");
      ProcessResult result3 = new ProcessResult("103", order , "100");
      DefaultResponseGenerator factory = new DefaultResponseGenerator();
      FrenchResponseGenerator frenchFactory = new FrenchResponseGenerator();
      Response resp1 = factory.prepareResponse(result1, result1.response());
      Response resp2 = frenchFactory.prepareResponse(result2, result2.response());
      Response resp3 = frenchFactory.prepareResponse(result3, result3.response());
   }
}
```

Earlier, the response generator referred to a factory internally, now the factory is exposed to the end user.

subclasses decide the type of response to be prepared

generate response in french dialect

```
extract and store data from process result
process success information
conversion not required
convert response to json format
```

Default Success Response

```
extract and store data from process result
process failure information
convert response to french dialect
convert response to json format
```

French Dialect Failure Response

```
extract and store data from process result
process data before formatting
convert response to french dialect
convert response to json format
```

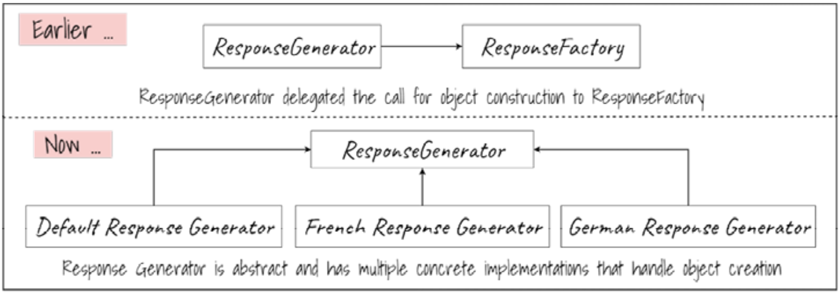
French Dialect TimeOut Response

***Figure 3.19**: Response factory tester II | Factory method pattern*

Earlier, the response generator referred to a factory internally, now the factory is exposed to the end user.

Factory method pattern

Let us now take a close look at the changes we have made and the working of the new design. In *Figure 3.20*, the first image depicts the changes, and the second one depicts the overall working of the design:

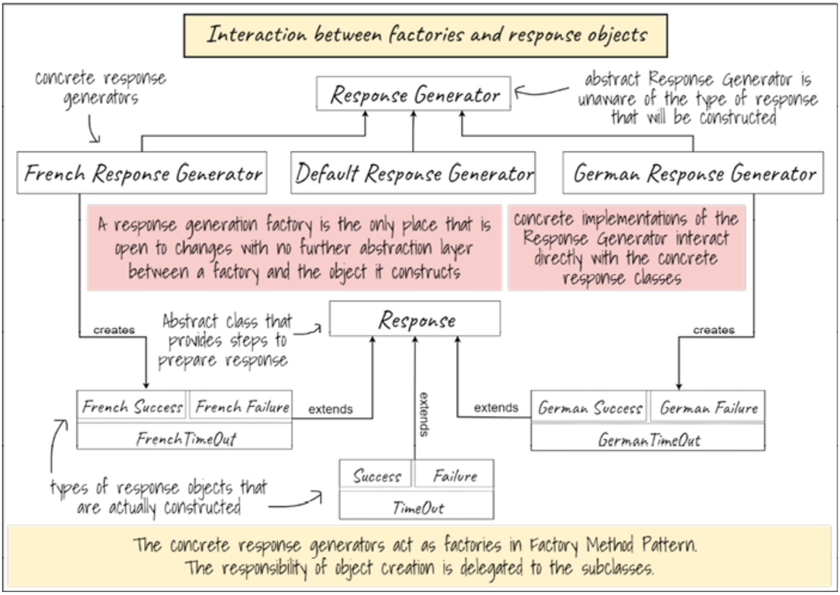

Earlier ...
ResponseGenerator
ResponseFactory
Now ...
ResponseGenerator
Default Response Generator
French Response Generator
German Response Generator

ResponseGenerator delegated the call for object construction to ResponseFactory

Response Generator is abstract and has multiple concrete implementations that handle object creation

***Figure 3.20**: Factory method pattern I*

Figure 3.21 exhibits the interaction between the concrete factories and the respective response types:


Interaction between factories and response objects
concrete response generators
Response Generator
French Response Generator
Default Response Generator
German Response Generator
creates
creates
Response
French Success
French Failure
FrenchTimeOut
extends
extends
extends
German Success
German Failure
GermanTimeOut
Success
Failure
TimeOut

abstract Response Generator is unaware of the type of response that will be constructed

A response generation factory is the only place that is open to changes with no further abstraction layer between a factory and the object it constructs

concrete implementations of the Response Generator interact directly with the concrete response classes

Abstract class that provides steps to prepare response

types of response objects that are actually constructed

The concrete response generators act as factories in Factory Method Pattern. The responsibility of object creation is delegated to the subclasses.

***Figure 3.21**: Factory method pattern II*

The factory method pattern, as described in *Figure 3.22*, provides an interface to construct an object but delegates the responsibility of construction to subclasses. The subclasses decide the class that should be instantiated.

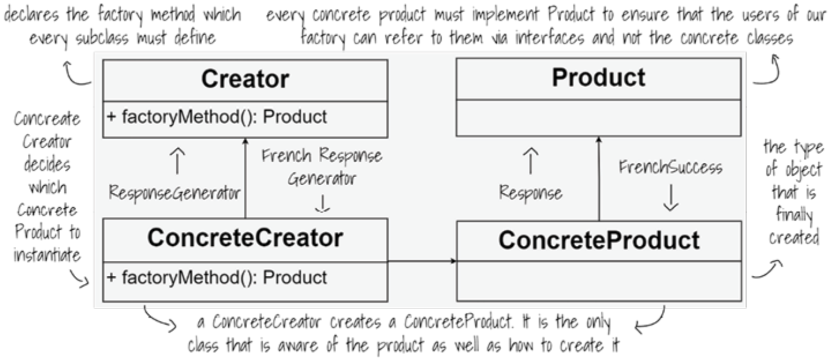
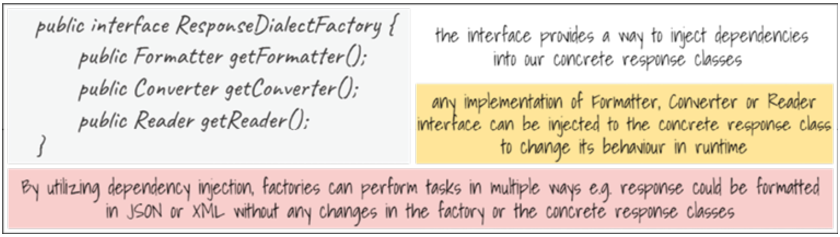

declares the factory method which every subclass must define
every concrete product must implement Product to ensure that the users of our factory can refer to them via interfaces and not the concrete classes
Creator
+ factoryMethod(): Product
Product
Concreate Creator decides which Concrete Product to instantiate
ResponseGenerator
French Response Generator
Response
FrenchSuccess
the type of object that is finally created
ConcreteCreator
+ factoryMethod(): Product
ConcreteProduct
a ConcreteCreator creates a ConcreteProduct. It is the only class that is aware of the product as well as how to create it

***Figure 3.22**: Factory method pattern defined*

The **new** operator is deemed harmful as it always constructs an object, and there is no encapsulation or abstraction involved. The factory method pattern encapsulates the process of object creation and makes it abstract for the caller; the caller does not know the type of object until it is returned. The factory method pattern can return the same object multiple times, or it can return a subtype of the type of object it creates. The **Creator** class knows nothing about the type of object that will be created; it depends on the subclass or concrete implementation of the creator used.

Family of objects

The *Report Generation System* now supports multiple factories and can be used to construct different responses. However, the responses are all created alike, as the factories support only a single way to read, convert, and format data to prepare the response. To use xml instead of Json to format the response, a new factory must be created, and that defeats the purpose of reusability. Refer to the following figure for a better understanding:



```
public interface ResponseDialectFactory {
        public Formatter getFormatter();
        public Converter getConverter();
        public Reader getReader();
}
```

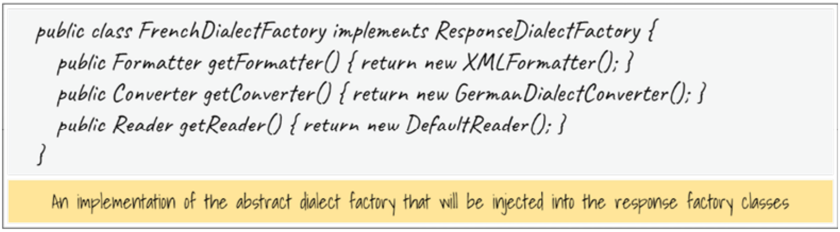
the interface provides a way to inject dependencies into our concrete response classes

any implementation of Formatter, Converter or Reader interface can be injected to the concrete response class to change its behaviour in runtime

By utilizing dependency injection, factories can perform tasks in multiple ways e.g. response could be formatted in JSON or XML without any changes in the factory or the concrete response classes

***Figure 3.23**: Response dialect factory | Abstract factory*

An interface, just like in *Figure 3.23*, can be used to inject the desired type of formatter, converter, and reader into concrete response classes to make them dynamic in nature and prepare the response in a variety of different ways:

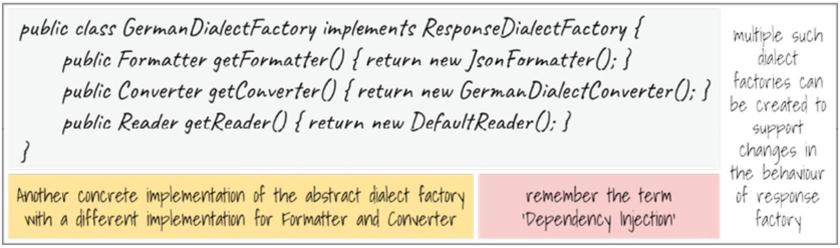


```
public class FrenchDialectFactory implements ResponseDialectFactory {
   public Formatter getFormatter() { return new XMLFormatter(); }
   public Converter getConverter() { return new GermanDialectConverter(); }
   public Reader getReader() { return new DefaultReader(); }
}
```

An implementation of the abstract dialect factory that will be injected into the response factory classes

Figure 3.24: *French dialect factory | Abstract factory*

Figure 3.24 and *Figure 3.25* demonstrate the concrete implementations of the response dialect factory interface. A concrete implementation of this interface can be injected into the concrete response classes.



```
public class GermanDialectFactory implements ResponseDialectFactory {
     public Formatter getFormatter() { return new JsonFormatter(); }
     public Converter getConverter() { return new GermanDialectConverter(); }
     public Reader getReader() { return new DefaultReader(); }
}
```

Another concrete implementation of the abstract dialect factory with a different implementation for Formatter and Converter

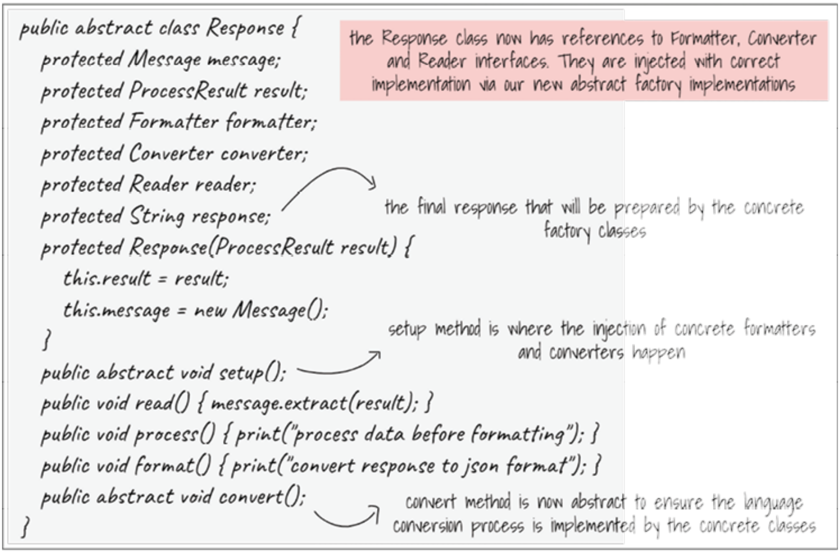
remember the term 'Dependency Injection'

multiple such dialect factories can be created to support changes in the behaviour of response factory

Figure 3.25: *German dialect Factory | Abstract factory*

This will introduce abstraction in the response classes, and the final response will then depend on the type of response dialect factory injected into the concrete response class.

Figure 3.26 exhibits an updated response class that includes references to the reader, converter, and formatter interfaces:



```
public abstract class Response {
   protected Message message;
   protected ProcessResult result;
   protected Formatter formatter;
   protected Converter converter;
   protected Reader reader;
   protected String response;
   protected Response(ProcessResult result) {
      this.result = result;
      this.message = new Message();
   }
   public abstract void setup();
   public void read() { message.extract(result); }
   public void process() { print("process data before formatting"); }
   public void format() { print("convert response to json format"); }
   public abstract void convert();
}
```

the Response class now has references to Formatter, Converter and Reader interfaces. They are injected with correct implementation via our new abstract factory implementations

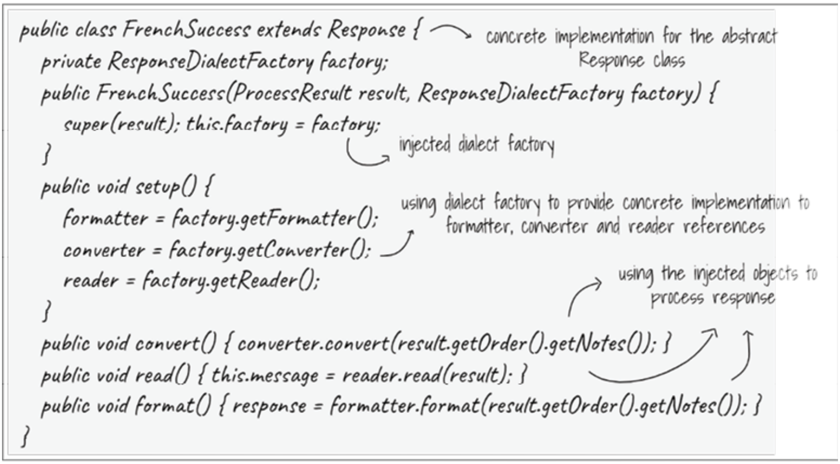
the final response that will be prepared by the concrete factory classes

setup method is where the injection of concrete formatters and converters happen

convert method is now abstract to ensure the language conversion process is implemented by the concrete classes

***Figure 3.26**: Abstract response | Abstract factory*

These methods would be used in the concrete response classes to prepare the desired response. A new abstract method **setup** is introduced, which must be implemented by the concrete implementations to configure the references added to the parent response class, as exhibited in *Figure 3.27*:



```
public class FrenchSuccess extends Response {
   private ResponseDialectFactory factory;
   public FrenchSuccess(ProcessResult result, ResponseDialectFactory factory) {
      super(result); this.factory = factory;
   }
   public void setup() {
      formatter = factory.getFormatter();
      converter = factory.getConverter();
      reader = factory.getReader();
   }
   public void convert() { converter.convert(result.getOrder().getNotes()); }
   public void read() { this.message = reader.read(result); }
   public void format() { response = formatter.format(result.getOrder().getNotes()); }
}
```

concrete implementation for the abstract Response class

injected dialect factory

using dialect factory to provide concrete implementation to formatter, converter and reader references

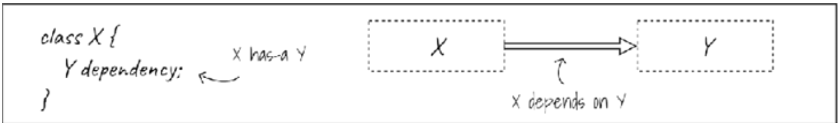
using the injected objects to process response

***Figure 3.27**: Concrete response French success | Abstract factory*

Dependency injection

Dependency injection is a concept that involves objects participating in a has-a relationship. It states that an object can be injected into another object while keeping both unaware of each other and developing independently. Dependency injection allows the design to be loosely coupled and configurable. A subtype of the actual type can also be injected in its place following the principles of polymorphism. It also reduces the effort required to test an application.

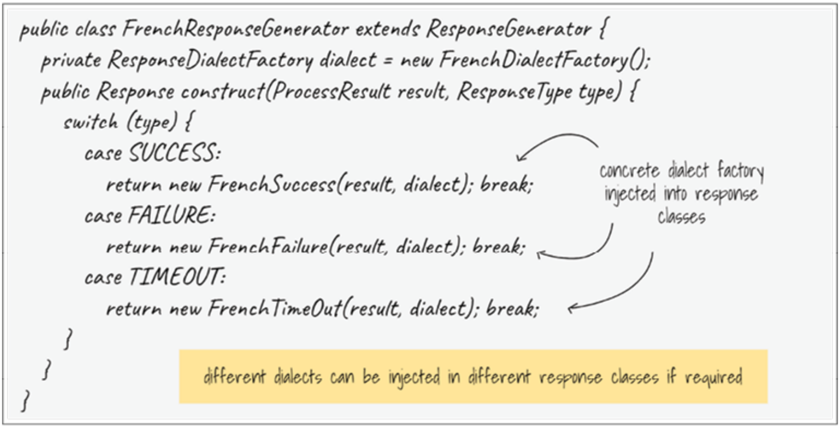
An object could be injected into another object either through the constructor or through a method of the dependent class. By using dependency injection, the creation and binding of the dependent objects can be moved outside of the class that depends on them, as shown in the following figure:



```
class X {
   Y dependency;
}
```

***Figure 3.28**: Dependency injection | Abstract factory*

In *Figure 3.29*, the class has an object of a dialect factory that is injected into the concrete response implementation. The response generator instance could itself be injected with an instance of a dialect factory, which could later be passed to the subclasses of the response class.



```
public class FrenchResponseGenerator extends ResponseGenerator {
   private ResponseDialectFactory dialect = new FrenchDialectFactory();
   public Response construct(ProcessResult result, ResponseType type) {
      switch (type) {
         case SUCCESS:
            return new FrenchSuccess(result, dialect); break;
         case FAILURE:
            return new FrenchFailure(result, dialect); break;
         case TIMEOUT:
            return new FrenchTimeOut(result, dialect); break;
      }
   }
}
```

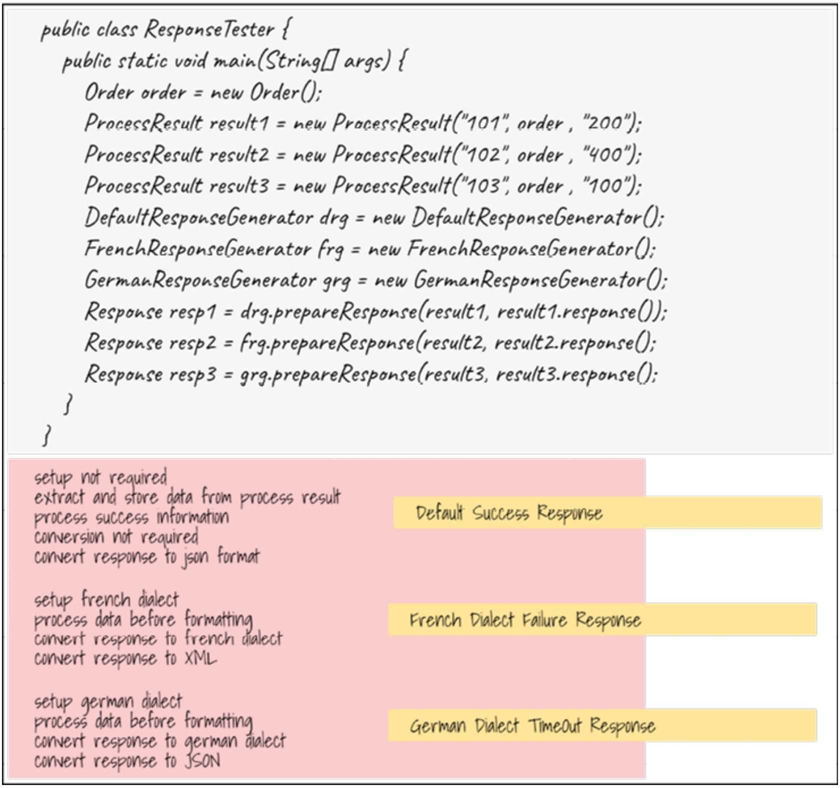
concrete dialect factory injected into response classes

different dialects can be injected in different response classes if required

***Figure 3.29**: French response generator | Abstract factory*

Response factory's third run

Let us do a test run to verify the changes we have made so far and to finally understand the abstract factory pattern as shown in *Figure 3.30*:



```
public class ResponseTester {
   public static void main(String[] args) {
      Order order = new Order();
      ProcessResult result1 = new ProcessResult("101", order , "200");
      ProcessResult result2 = new ProcessResult("102", order , "400");
      ProcessResult result3 = new ProcessResult("103", order , "100");
      DefaultResponseGenerator drg = new DefaultResponseGenerator();
      FrenchResponseGenerator frg = new FrenchResponseGenerator();
      GermanResponseGenerator grg = new GermanResponseGenerator();
      Response resp1 = drg.prepareResponse(result1, result1.response());
      Response resp2 = frg.prepareResponse(result2, result2.response();
      Response resp3 = grg.prepareResponse(result3, result3.response();
   }
}
```

```
setup not required
extract and store data from process result
process success information                    Default Success Response
conversion not required
convert response to json format

setup french dialect
process data before formatting                 French Dialect Failure Response
convert response to french dialect
convert response to XML

setup german dialect
process data before formatting                 German Dialect TimeOut Response
convert response to german dialect
convert response to JSON
```

Figure 3.30: *Response factory tester III | Abstract factory*

Abstract factory

The abstract factory shown in *Figure 3.31* provides an interface to create a family of interrelated or dependent objects without specifying their concrete classes. The concrete factories implement various product families and allow the factory to instantiate any of the product objects.

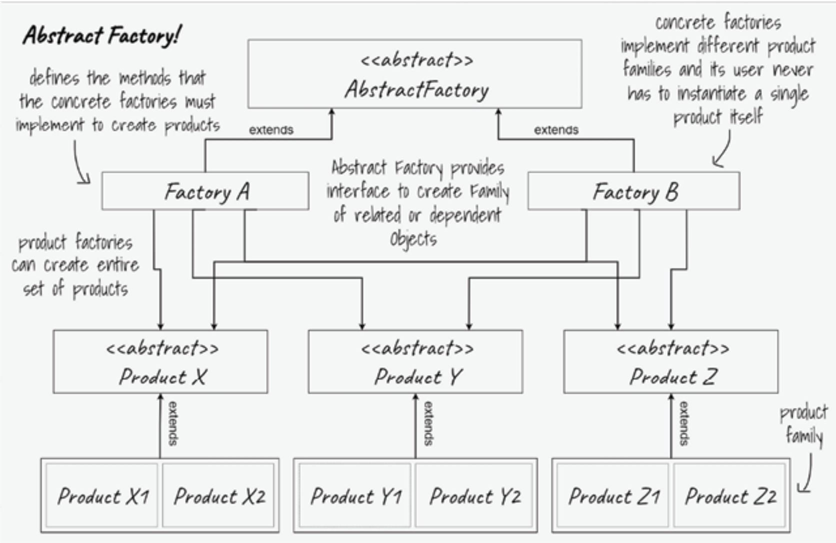

Abstract Factory!
defines the methods that the concrete factories must implement to create products
<<abstract>>
AbstractFactory
concrete factories implement different product families and its user never has to instantiate a single product itself
extends
extends
Factory A
Abstract Factory provides interface to create Family of related or dependent Objects
Factory B
product factories can create entire set of products
<<abstract>>
Product X
<<abstract>>
Product Y
<<abstract>>
Product Z
extends
extends
extends
product family
Product X1
Product X2
Product Y1
Product Y2
Product Z1
Product Z2

***Figure 3.31**: Abstract factory defined*

Figure 3.32 exhibits the abstract factory implementation used in our response generator system:

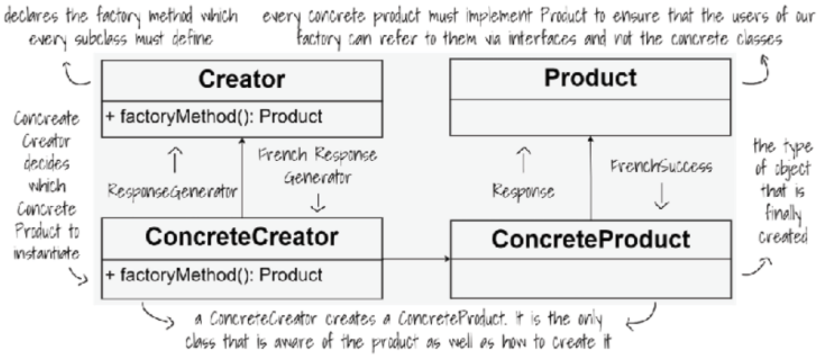

declares the factory method which every subclass must define
every concrete product must implement Product to ensure that the users of our factory can refer to them via interfaces and not the concrete classes
Creator
+ factoryMethod(): Product
Product
Concreate Creator decides which Concrete Product to instantiate
ResponseGenerator
French Response Generator
Response
FrenchSuccess
the type of object that is finally created
ConcreteCreator
+ factoryMethod(): Product
ConcreteProduct
a ConcreteCreator creates a ConcreteProduct. It is the only class that is aware of the product as well as how to create it

***Figure 3.32**: Abstract factory interaction*

Advantages and drawbacks

The family of factory design patterns has many advantages over the direct use of constructors for creating objects. Let us discuss the benefits first, followed by the drawbacks:

- It promotes loose coupling by using interfaces for interaction between classes.

- It encapsulates object creation logic in a separate class, allowing for easy maintenance and modification.
- It provides a centralized approach to object creation, which eases testing and maintenance.
- It promotes code reusability by centralizing the object creation logic.
- It supports the open/closed principle and allows for the addition of new objects to the factory without modifying existing code.

The drawbacks of the builder design pattern are as follows:

- It can increase complexity by introducing additional interfaces and supporting classes, especially when there are multiple factories to deal with, as in the case of using an abstract factory implementation.
- Although it promotes loose coupling, the factory classes may be tightly coupled to the concrete classes that they interact with.
- It can impact the performance of an application if the factories are not properly implemented and if they manage the construction of many object types.

Usage

Some of the scenarios relevant to the family of factory design patterns are as follows:

- It can be used for creating objects at runtime and when the object types are not known in advance, and when it is decided by the logical flow of the application.
- It can be used to encapsulate and centralize the object creation process to reduce coupling. This makes it easier to make changes in the factory implementation without affecting the end user.

Conclusion

In this chapter, we learned that factory design patterns are creational patterns. They hide the complexity of object creation from the user by providing an interface that can create objects for them. It encapsulates object creation, promotes loose coupling by interacting with abstract types rather than concrete ones, and drives towards a flexible design approach. Factory patterns can also increase the complexity of the code if not used properly, and should be avoided if techniques like a simple factory are sufficient. They support the open/closed principle, which means classes should be closed for modification but open for extension, and the dependency inversion principle, which states that code should depend on abstractions rather than concrete classes. We also learned that the factory method pattern provides an interface for object creation but delegates the responsibility of creation to the subclasses. The abstract factory pattern provides an interface to create a family of objects without specifying the concrete classes.

In the next chapter, we will discuss the builder design pattern and learn about decoupling object creation from its representation.

Questions

In this section, we will sharpen our brains by indulging in some fun brain activities.

Find the odd one out

1. Simple factory, abstract factory, factory method, virtual factory
2. Family of objects, dependency injection, X has-a Y, new keyword
3. Abstraction, interface, encapsulation, concrete class
4. Dependency injection:
   a. Involves two objects
   b. Support polymorphic types
   c. Can be done for unrelated objects
   d. Require composition
5. Abstract factory:

    a. Promotes loose coupling
    b. Require inheritance
    c. Creates a family of related objects
    d. Can only support a handful of classes

Select two correct answers

1. Factory method pattern:
    a. Delegates the responsibility of object creation to subclasses
    b. Promotes the open/close principle
    c. Does not support subclass types as return values
    d. Can only take one decision parameter as input
2. Abstract factory:
    a. Can only have static methods
    b. Promotes abstraction
    c. Supports dependency injection
    d. Is a factory of abstract classes
3. Interfaces:
    a. Are the pillars of factory patterns
    b. Can only be implemented by other interfaces
    c. Can be replaced with abstract classes in an abstract factory
    d. Support dependency injection
4. Loose coupling can be achieved by utilizing:
    a. Interfaces
    b. Subclasses
    c. Tight coupling
    d. Dependency injection

Answers

This section provides the answers to the questions given.

Find the odd one out

1. Virtual factory
2. New keyword
3. Encapsulation
4. Can be done for unrelated objects
5. Can only support a handful of classes

Select two correct answers

1. Delegates the responsibility of object creation to subclasses, promotes the open/close principle
2. Promotes abstraction, supports dependency injection
3. Are the pillars of factory pattern, can be replaced with abstract classes in an abstract factory
4. Interface, dependency injection

Join our Discord space

Join our Discord workspace for latest updates, offers, tech happenings around the world, new releases, and sessions with the authors:

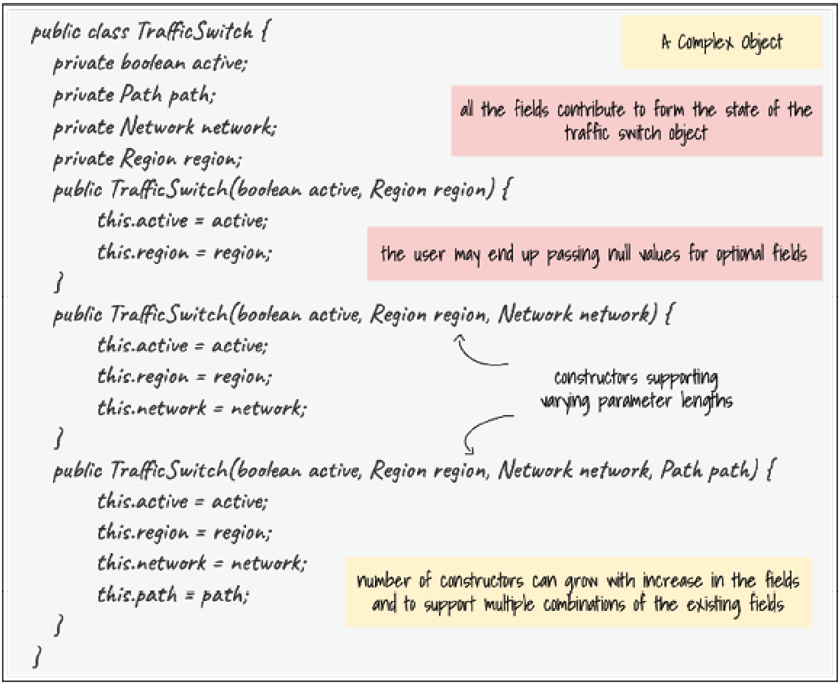
https://discord.bpbonline.com

CHAPTER 4

Delegate Object Construction

Introduction

An object's state is critical for an application to function properly. An incomplete or inconsistent state can be a cause of exceptions and faulty behavior. An object's state is defined in the form of data members it contains and the possible values it can have. *Figure 4.1* exhibits a complex object and a few of its constructors. An object can have many constructors depending on the number of data members and its possible representations.



```
public class TrafficSwitch {
    private boolean active;
    private Path path;
    private Network network;
    private Region region;
    public TrafficSwitch(boolean active, Region region) {
        this.active = active;
        this.region = region;
    }
    public TrafficSwitch(boolean active, Region region, Network network) {
        this.active = active;
        this.region = region;
        this.network = network;
    }
    public TrafficSwitch(boolean active, Region region, Network network, Path path) {
        this.active = active;
        this.region = region;
        this.network = network;
        this.path = path;
    }
}
```

A Complex Object

all the fields contribute to form the state of the traffic switch object

the user may end up passing null values for optional fields

constructors supporting varying parameter lengths

number of constructors can grow with increase in the fields and to support multiple combinations of the existing fields

***Figure 4.1**: Class with multiple constructors*

The data members are directly related to the complexity of an object. A simple object has only a few data members, whereas a complex object can have many data members, each contributing to the increase in complexity.

It is often straightforward to construct a simple object using the **new** keyword, where the number of fields is limited, for example, a string or an integer. In the case of a complex object, there can be multiple constructors for various representations of an object's state. Selecting the correct constructor for such a class can be confusing and can lead to errors due to uninstantiated data members.

Though one may contradict, saying that one single constructor containing all the fields is sufficient and can take care of all the scenarios, this approach has a few drawbacks. The first drawback is that a lot of **null** values can be passed to the constructor, mistaking some fields to be optional. The second drawback is that it requires a complex validation mechanism to confirm the correctness of the passed values, which is somehow easier to manage with the multiple constructor approach or the telescoping approach.

Structure

This chapter covers the following topics:

- Delegating object construction
- Implementing builders
- Summary
- Advantages and drawbacks
- Usage

Objectives

By the end of this chapter, the readers will learn about a modular approach to building complex objects that have multiple representations. Objects with many data members usually expose multiple constructors, and working with such objects can be error-prone if they are constructed using an incorrect constructor. The builder design pattern provides a structured mechanism to build an object that ensures the instantiation of the required data members before their use.

After finishing this chapter, they will be able to understand the workings of the builder design pattern and how it can be used to construct complex objects with ease and efficiency.

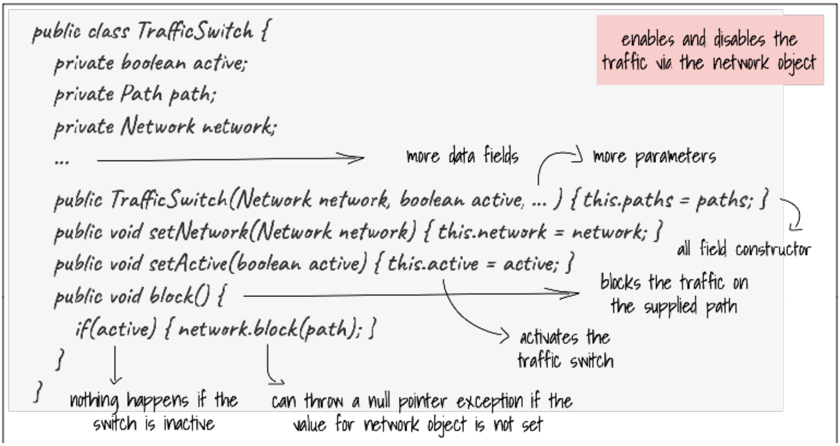
Delegating object construction

The state of an object is a serious affair; it can impact the behavior of that object, resulting in an unexpected output or even exceptions. An object's state can be manipulated using setter methods that can set a specific value to a data field.

Another way to construct complex objects is to use an approach that utilizes methods to build the state of an object one step at a time. In this approach, the user does not directly interact with the constructor of the class; instead, the task of constructing the object is achieved via method calls. Each method constructs only a part of the object state and not the entire object. Multiple such methods can be chained together to form the desired state of the object before finally calling the build method that returns the object itself.

Construction of objects can also be delegated to other classes. A class that takes up the responsibility to construct a single type of object, validates and verifies the correctness of the input, and returns an object with a consistent state. The object formed in the process could also be immutable if required.

One can argue that by combining the setter methods, more commonly known as the setters, with the all-field constructor approach, the desired state can be achieved at any given time. Technically, that is a valid statement if the setters are provided for all the fields, and one does not fail to call them. The problem with that approach is that the state of the overall object is always dangling between consistent and inconsistent, and its behavior cannot be guaranteed. If the values for the object fields are not set properly, the behavior is unpredictable, and hence, it is always on the developer to ensure that the relevant setters are called and to ensure that runtime exceptions are not thrown away due to an improper state of the object. Refer to the following figure for a better understanding:



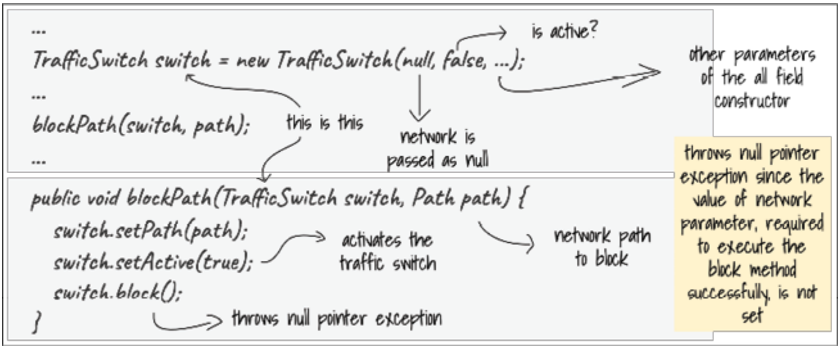
```
public class TrafficSwitch {
  private boolean active;
  private Path path;
  private Network network;
  ...
  public TrafficSwitch(Network network, boolean active, ... ) { this.paths = paths; }
  public void setNetwork(Network network) { this.network = network; }
  public void setActive(boolean active) { this.active = active; }
  public void block() {
    if(active) { network.block(path); }
  }
}
```

enables and disables the traffic via the network object
more data fields
more parameters
all field constructor
blocks the traffic on the supplied path
activates the traffic switch
nothing happens if the switch is inactive
can throw a null pointer exception if the value for network object is not set

***Figure 4.2**: Setters and all field constructors*

Let us go through an example to understand and evaluate the benefits of the approach discussed in this section. The class in *Figure 4.2* is a refactored variant of the complex class in *Figure 4.1*. For the **block** method to produce the expected result, it is required that the object is in the active state, that is, the active field is set to true, and that it has a valid reference to a network instance.

In *Figure 4.3*, the value of a network is initially passed as **null**, and later it was missed to be set before calling the **block** method:



```
...
TrafficSwitch switch = new TrafficSwitch(null, false, ...);
...
blockPath(switch, path);
...
```

```
public void blockPath(TrafficSwitch switch, Path path) {
   switch.setPath(path);
   switch.setActive(true);
   switch.block();
}
```

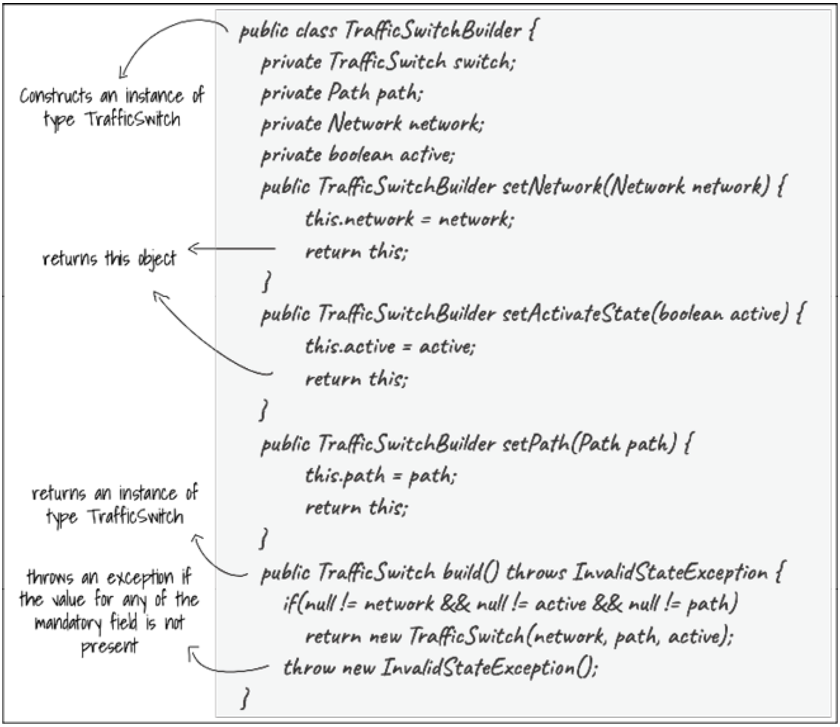
is active?
other parameters of the all field constructor
this is this
network is passed as null
throws null pointer exception since the value of network parameter, required to execute the block method successfully, is not set
activates the traffic switch
network path to block
throws null pointer exception

***Figure 4.3**: Block traffic*

The **blockPath** method throws a **NullPointerException** because the state of the object is inconsistent. It is important to understand that the object should be properly constructed with all the necessary fields to mitigate the chances of erratic behavior.

Let us revisit our discussion on the delegation of the object construction process and view a sample class that can validate and construct objects for us while ensuring the consistency of the finally constructed object.

Figure 4.4 exhibits the use of the builder design pattern to construct an object by delegating the responsibility of object construction to a builder class:



```
public class TrafficSwitchBuilder {
  private TrafficSwitch switch;
  private Path path;
  private Network network;
  private boolean active;
  public TrafficSwitchBuilder setNetwork(Network network) {
      this.network = network;
      return this;
  }
  public TrafficSwitchBuilder setActivateState(boolean active) {
      this.active = active;
      return this;
  }
  public TrafficSwitchBuilder setPath(Path path) {
      this.path = path;
      return this;
  }
  public TrafficSwitch build() throws InvalidStateException {
    if(null != network && null != active && null != path)
      return new TrafficSwitch(network, path, active);
    throw new InvalidStateException();
}
```

Constructs an instance of type TrafficSwitch

returns this object

returns an instance of type TrafficSwitch

throws an exception if the value for any of the mandatory field is not present

***Figure 4.4**: Traffic switch builder*

Representational state

Constructors provide us the liberty to construct object instances with varying representational states, and a corresponding behavior change. Complex objects having many data members or those that are composed of other objects often have multiple constructors with different sets of input parameters. All these parameters, in various combinations, represent a different state of the object, and we often see constructors corresponding to these representations.

Though using a constructor is probably the easiest of the approaches available to construct an object, it is certainly not the most readable, especially when the constructor parameters begin to grow. As a user of that class, it is hard to figure out which constructor represents which exact state of the object, and we often tend to pass in **null** values for one or more of the optional parameters while constructing an object.

As discussed in the preceding section and demonstrated in *Figure*

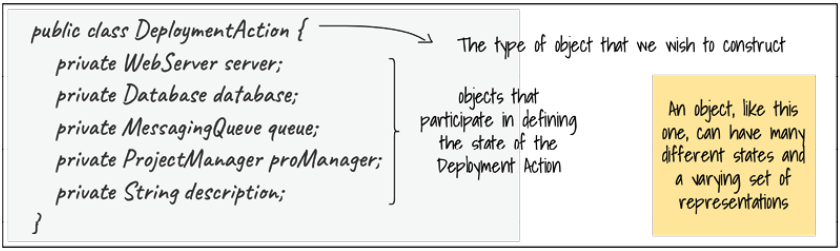
4.4, we could delegate the responsibility of object creation to another class that exposes methods to set each of the data members and a **build** method to finally construct an object of the actual type. It can also validate the mandatory parameters before constructing a valid representation of that object. There could be multiple variations of the build method to produce different representations of the same object.

In the following section, we will discuss the benefits and drawbacks of delegating the task of constructing an object to another class and will go through an example, step by step, to analyze and understand the design of the builder class. We will also discuss some of the variations of our design and will observe how that impacts the overall architecture.

Implementing builders

This example uses builders to construct objects. Imagine that we have taken up this huge task of setting up an automated process to deploy virtual machines on our cloud infrastructure as and when requested by our customers. This requires us to set up a service that takes the input from our customers via a web page and then processes it to deploy a virtual machine.

As demonstrated in *Figure 4.5*, the class stores the information related to the strategy selected by our customer to deploy a virtual machine on our cloud infrastructure, and the necessary software along with it:



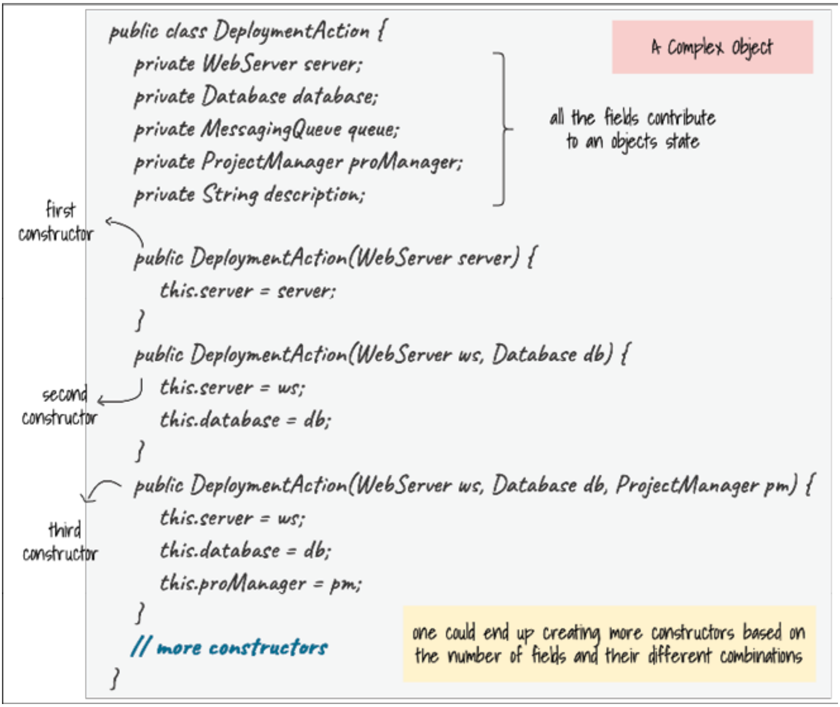
```
public class DeploymentAction {
    private WebServer server;
    private Database database;
    private MessagingQueue queue;
    private ProjectManager proManager;
    private String description;
}
```

The type of object that we wish to construct
objects that participate in defining the state of the Deployment Action
An object, like this one, can have many different states and a varying set of representations

***Figure 4.5**: Deployment action*

There could be multiple variations of the deployment strategy, and each strategy might require a different set of software. We must ensure that the constructed **DeploymentAction** object is populated with the right software mix that shall be installed on the virtual machine.

One of the ways to support this behavior is to have multiple constructors that support various combinations. *Figure 4.6* demonstrates a class with multiple constructors:

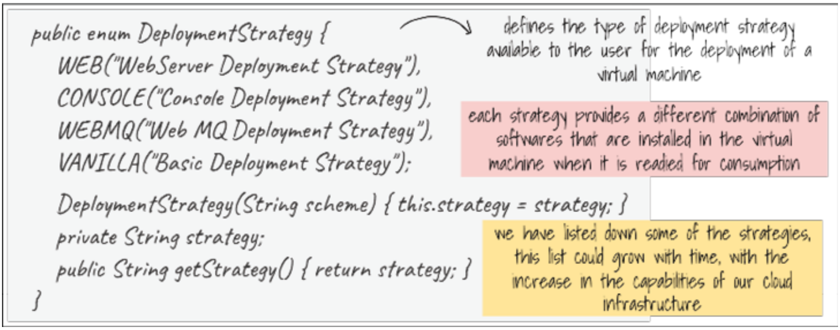


```
public class DeploymentAction {
    private WebServer server;
    private Database database;
    private MessagingQueue queue;
    private ProjectManager proManager;
    private String description;

    public DeploymentAction(WebServer server) {
        this.server = server;
    }
    public DeploymentAction(WebServer ws, Database db) {
        this.server = ws;
        this.database = db;
    }
    public DeploymentAction(WebServer ws, Database db, ProjectManager pm) {
        this.server = ws;
        this.database = db;
        this.proManager = pm;
    }
    // more constructors
}
```

A Complex Object
all the fields contribute to an objects state
first constructor
second constructor
third constructor
one could end up creating more constructors based on the number of fields and their different combinations

***Figure 4.6**: Multiple constructors*

The API that is exposed to the customers requires a deployment strategy for setting up virtual machines on the cloud. To simplify the selection process, we have created an enum that declares all the available strategies supported by the system. *Figure 4.7* exhibits the enum that exposes these strategies:



```
public enum DeploymentStrategy {
    WEB("WebServer Deployment Strategy"),
    CONSOLE("Console Deployment Strategy"),
    WEBMQ("Web MQ Deployment Strategy"),
    VANILLA("Basic Deployment Strategy");

    DeploymentStrategy(String scheme) { this.strategy = strategy; }
    private String strategy;
    public String getStrategy() { return strategy; }
}
```

defines the type of deployment strategy available to the user for the deployment of a virtual machine

each strategy provides a different combination of softwares that are installed in the virtual machine when it is readied for consumption

we have listed down some of the strategies, this list could grow with time, with the increase in the capabilities of our cloud infrastructure

***Figure 4.7**: Deployment strategy*

An entry can be added to **DeploymentStrategy** whenever a new strategy is supported by the system and is made available to the customer.

We can use the constructed object to deploy and configure the virtual machine with the necessary software. This solution is fine for cases where there are fewer combinations of input parameters. In our case, there could be many different variations possible, and the direct use of constructors might end up being too complex and less readable. Another way to construct an object of type **DeploymentAction** is to make use of a similar design demonstrated in *Figure 4.4*.

The builder design pattern aims to build an object in a stepwise manner that separates the process of construction of an object from its representation. It focuses on building an object with a complete state with all the required fields set up using method calls and eliminates the direct interaction with constructors.

Figure 4.8 exhibits the use of different builder implementations to build instances of the same class but with different representational states:

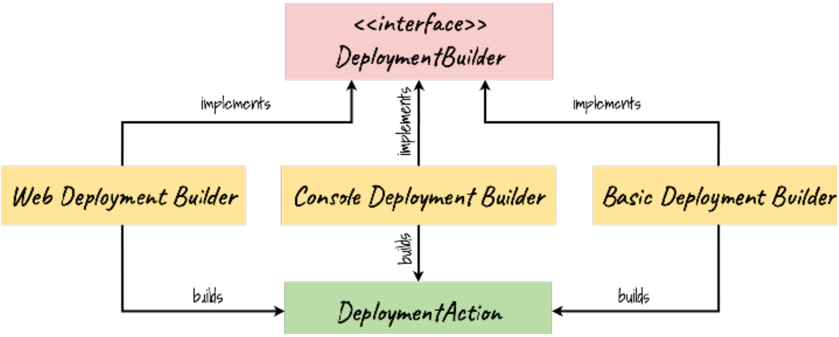
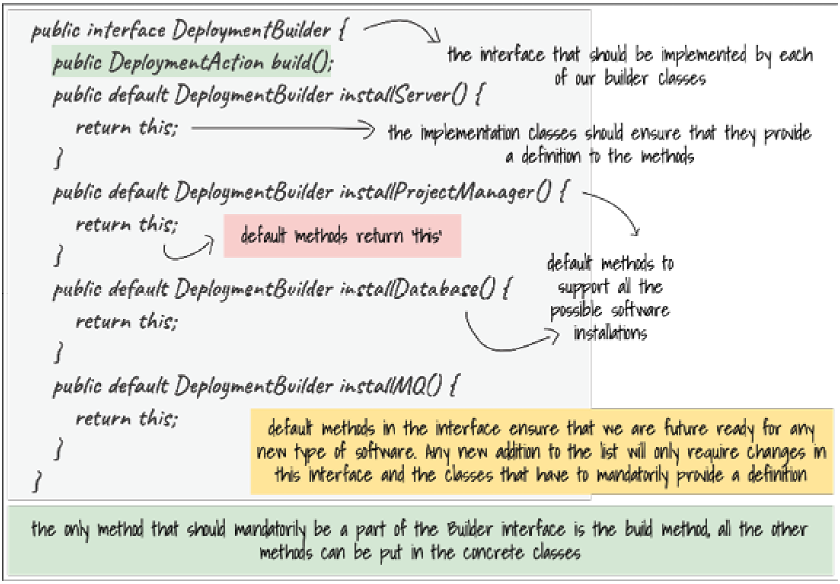

<<interface>>
DeploymentBuilder
implements
implements
implements
Web Deployment Builder
Console Deployment Builder
Basic Deployment Builder
builds
builds
builds
DeploymentAction

Figure 4.8: *Basic design*

There could be multiple representations of the same object. We can either create multiple builder classes for each representation type or construct a single builder class and expose different methods to support multiple representations. For this example, we will construct multiple deployment builders and will explore the other option later. To ensure that the different builder classes follow the same contract, we will first build an interface that will list down the behavior our builders should support.

The interface

The **DeploymentBuilder** interface defines the methods that should be supported by all the concrete builders. As you can see, only the build method is truly abstract, while the others are not. Other methods provide a default implementation, which is just to return the object itself. Since our concrete builders return different representations of the same object, they only require overriding those methods that are relevant to them, as shown in *Figure 4.9*:



```
public interface DeploymentBuilder {
    public DeploymentAction build();
    public default DeploymentBuilder installServer() {
        return this;
    }
    public default DeploymentBuilder installProjectManager() {
        return this;
    }
    public default DeploymentBuilder installDatabase() {
        return this;
    }
    public default DeploymentBuilder installMQ() {
        return this;
    }
}
```

the interface that should be implemented by each of our builder classes

the implementation classes should ensure that they provide a definition to the methods

default methods return 'this'

default methods to support all the possible software installations

default methods in the interface ensure that we are future ready for any new type of software. Any new addition to the list will only require changes in this interface and the classes that have to mandatorily provide a definition

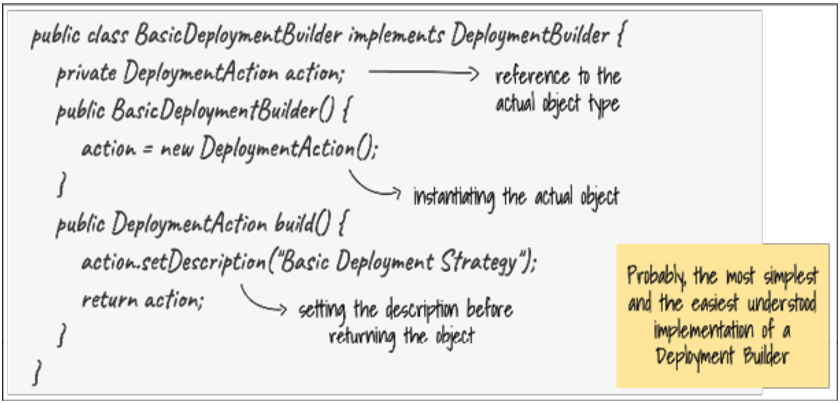
the only method that should mandatorily be a part of the Builder interface is the build method, all the other methods can be put in the concrete classes

***Figure 4.9**: Deployment builder interface*

One important aspect of this design is that the construction of the object is not complete until the build method is called on the builder. The methods can be chained together to prepare the object before it is finally constructed. This also means that there is only one way in which the object is exposed to the outside world. The **build** method can also be used to validate the subject's state for completeness before its construction, or throw an exception if the state of the object is incomplete. This will be discussed in detail in the following chapter.

Concrete builders

Figure 4.10 demonstrates the simplest implementation of a concrete builder. It is mapped to the default deployment strategy and does not support the installation of any type of software. It implements the builder interface and only overrides the abstract build method. It also does not perform any validation since the only mandatory parameter, that is, description, is always set to a value in the build method itself.



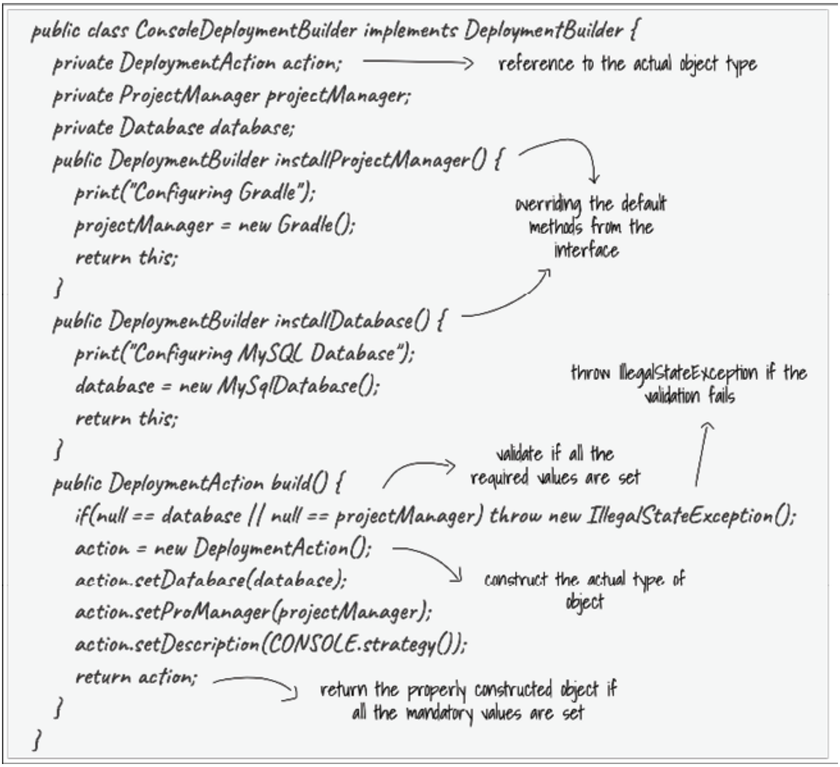
```
public class BasicDeploymentBuilder implements DeploymentBuilder {
   private DeploymentAction action;
   public BasicDeploymentBuilder() {
      action = new DeploymentAction();
   }
   public DeploymentAction build() {
      action.setDescription("Basic Deployment Strategy");
      return action;
   }
}
```

reference to the actual object type
instantiating the actual object
setting the description before returning the object
Probably, the most simplest and the easiest understood implementation of a Deployment Builder

***Figure 4.10**: Basic deployment builder*

Figure 4.11 exhibits another example of a concrete implementation of the builder interface. It configures the installation of *Gradle* as a project management tool and *MySQL* as the database engine. It also validates the completeness of the object before exposing it to the outside world. Notice that the builder class also contains the same references as that of the subject.

Each concrete builder builds a specific representation of the subject. Instead of selecting which constructor to call, we can just choose the specific representation and state we want to build. This is much more readable as each builder provides methods with names that are descriptive enough for the user to understand. The builders can also validate the state before finally constructing an object, something that is missing in the direct constructor approach. Refer to the following figure to build an understanding:



```
public class ConsoleDeploymentBuilder implements DeploymentBuilder {
  private DeploymentAction action;
  private ProjectManager projectManager;
  private Database database;
  public DeploymentBuilder installProjectManager() {
    print("Configuring Gradle");
    projectManager = new Gradle();
    return this;
  }
  public DeploymentBuilder installDatabase() {
    print("Configuring MySQL Database");
    database = new MySqlDatabase();
    return this;
  }
  public DeploymentAction build() {
    if(null == database || null == projectManager) throw new IllegalStateException();
    action = new DeploymentAction();
    action.setDatabase(database);
    action.setProManager(projectManager);
    action.setDescription(CONSOLE.strategy());
    return action;
  }
}
```

reference to the actual object type

overriding the default methods from the interface

throw IllegalStateException if the validation fails

validate if all the required values are set

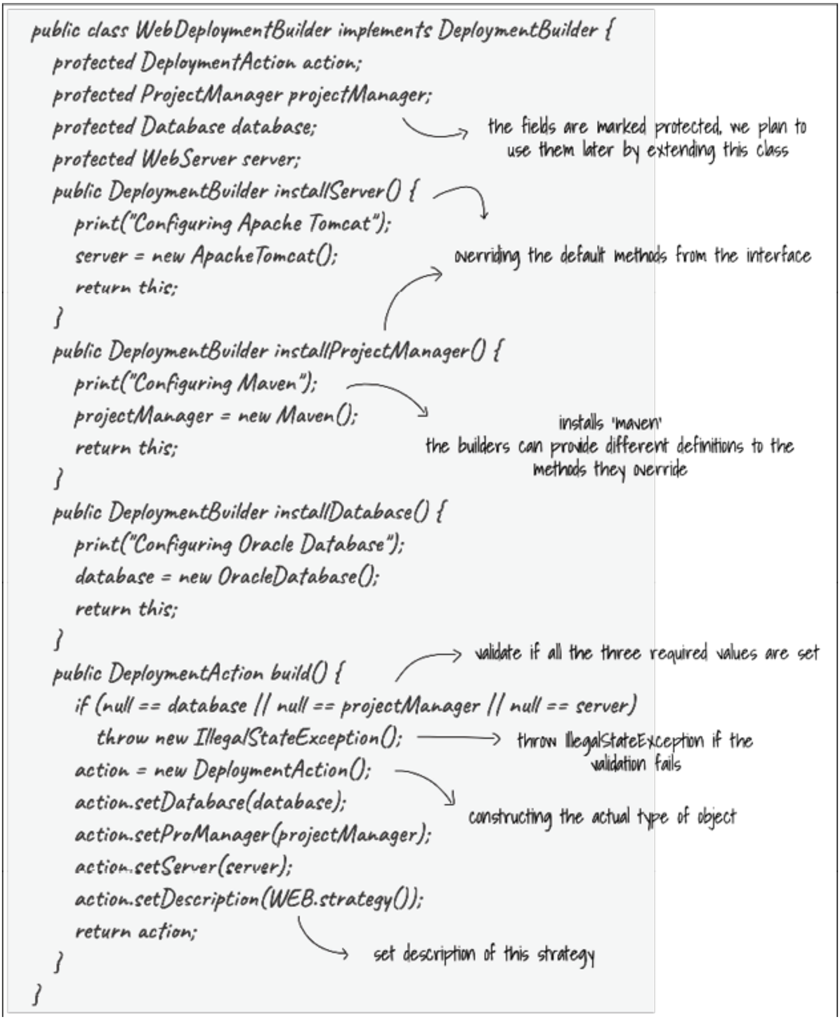
construct the actual type of object

return the properly constructed object if all the mandatory values are set

Figure 4.11: *Console deployment builder*

Figure 4.11 demonstrates another example of the deployment builder. It configures the installation of *Apache Tomcat* as an application server, *Maven* as a project management tool, and *Oracle* as the database engine. It also validates the completeness of the object by verifying that mandatory requirements are fulfilled before exposing it to the outside world.

In *Figure 4.12*, the data members are marked protected and not private. This is useful in defining similar deployment strategies with only minor changes. In such a scenario, this builder class can be extended with relevant sections overridden.



```
public class WebDeploymentBuilder implements DeploymentBuilder {
    protected DeploymentAction action;
    protected ProjectManager projectManager;
    protected Database database;
    protected WebServer server;
    public DeploymentBuilder installServer() {
        print("Configuring Apache Tomcat");
        server = new ApacheTomcat();
        return this;
    }
    public DeploymentBuilder installProjectManager() {
        print("Configuring Maven");
        projectManager = new Maven();
        return this;
    }
    public DeploymentBuilder installDatabase() {
        print("Configuring Oracle Database");
        database = new OracleDatabase();
        return this;
    }
    public DeploymentAction build() {
        if (null == database || null == projectManager || null == server)
            throw new IllegalStateException();
        action = new DeploymentAction();
        action.setDatabase(database);
        action.setProManager(projectManager);
        action.setServer(server);
        action.setDescription(WEB.strategy());
        return action;
    }
}
```

the fields are marked protected, we plan to use them later by extending this class

overriding the default methods from the interface

installs 'maven'

the builders can provide different definitions to the methods they override

validate if all the three required values are set

throw IllegalStateException if the validation fails

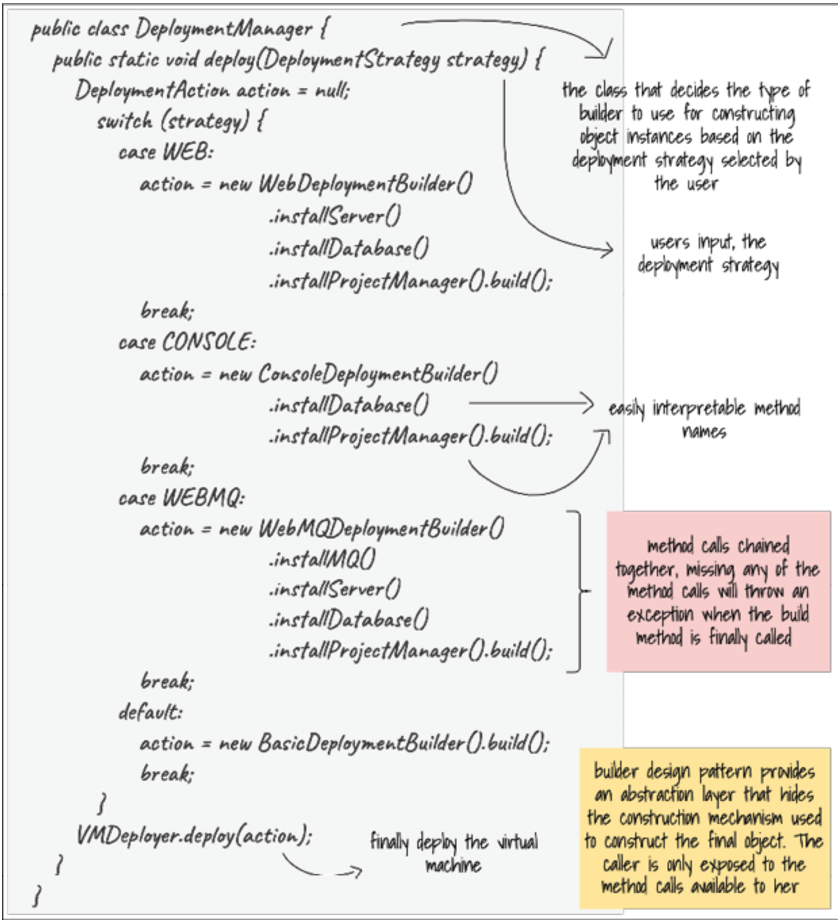
constructing the actual type of object

set description of this strategy

***Figure 4.12**: Web deployment builder*

Deployment manager

The class in *Figure 4.13* is responsible for selecting the right deployment builder to build the **Action** object, based on the strategy selected by the user:



```
public class DeploymentManager {
  public static void deploy(DeploymentStrategy strategy) {
    DeploymentAction action = null;
      switch (strategy) {
        case WEB:
          action = new WebDeploymentBuilder()
                          .installServer()
                          .installDatabase()
                          .installProjectManager().build();
          break;
        case CONSOLE:
          action = new ConsoleDeploymentBuilder()
                          .installDatabase()
                          .installProjectManager().build();
          break;
        case WEBMQ:
          action = new WebMQDeploymentBuilder()
                          .installMQ()
                          .installServer()
                          .installDatabase()
                          .installProjectManager().build();
          break;
        default:
          action = new BasicDeploymentBuilder().build();
          break;
      }
    VMDeployer.deploy(action);
  }
}
```

the class that decides the type of builder to use for constructing object instances based on the deployment strategy selected by the user

users input, the deployment strategy

easily interpretable method names

method calls chained together, missing any of the method calls will throw an exception when the build method is finally called

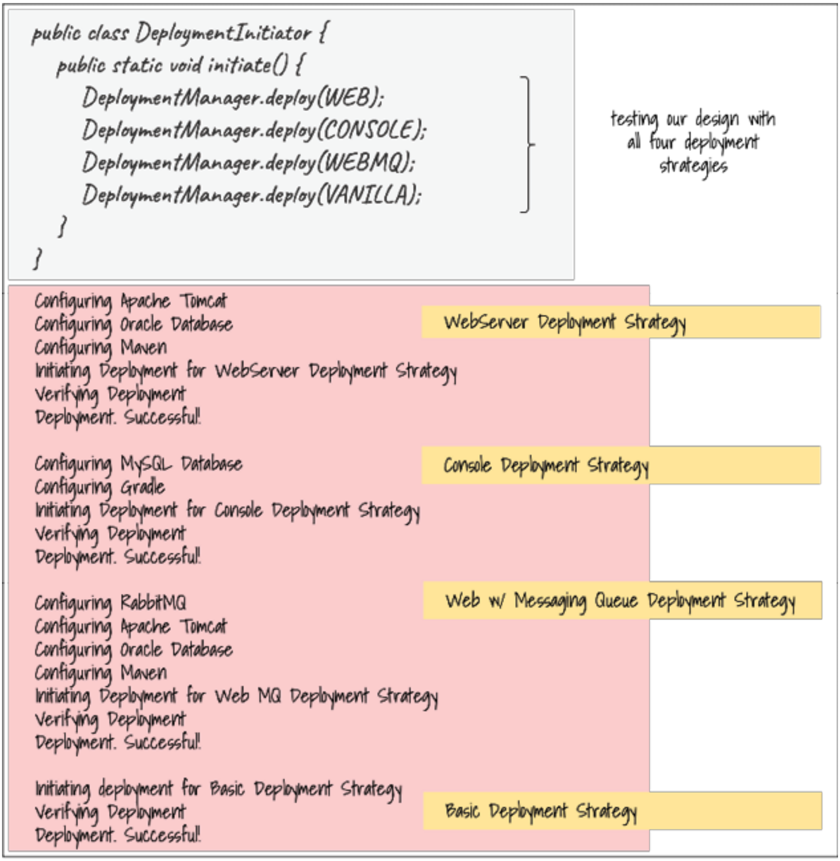
builder design pattern provides an abstraction layer that hides the construction mechanism used to construct the final object. The caller is only exposed to the method calls available to her

finally deploy the virtual machine

***Figure 4.13**: Deployment manager*

Testing the builders

We have written a test class to verify the working of the builder design pattern. Let us execute the same as shown in *Figure 4.14*:



```
public class DeploymentInitiator {
   public static void initiate() {
      DeploymentManager.deploy(WEB);
      DeploymentManager.deploy(CONSOLE);
      DeploymentManager.deploy(WEBMQ);
      DeploymentManager.deploy(VANILLA);
   }
}
```

```
Configuring Apache Tomcat
Configuring Oracle Database
Configuring Maven
Initiating Deployment for WebServer Deployment Strategy
Verifying Deployment
Deployment. Successful!

Configuring MySQL Database
Configuring Gradle
Initiating Deployment for Console Deployment Strategy
Verifying Deployment
Deployment. Successful!

Configuring RabbitMQ
Configuring Apache Tomcat
Configuring Oracle Database
Configuring Maven
Initiating Deployment for Web MQ Deployment Strategy
Verifying Deployment
Deployment. Successful!

Initiating deployment for Basic Deployment Strategy
Verifying Deployment
Deployment. Successful!
```

testing our design with all four deployment strategies
WebServer Deployment Strategy
Console Deployment Strategy
Web w/ Messaging Queue Deployment Strategy
Basic Deployment Strategy

***Figure 4.14**: Deployment initiator*

In *Figure 4.14*, multiple calls are made to the **deploy** method of **DeploymentManager**, passing along the strategy that should be used for deploying the virtual machine on the cloud infrastructure. The manager chooses the builder to build the correct representation of the object, which is then passed to another class for the final deployment.

Summary

The builder design pattern, as shown in *Figure 4.15*, is one of the most widely used creational design patterns that defines a phased object construction process, which can be reused to produce different representations of the same object. It separates the process of constructing an object from its representation.

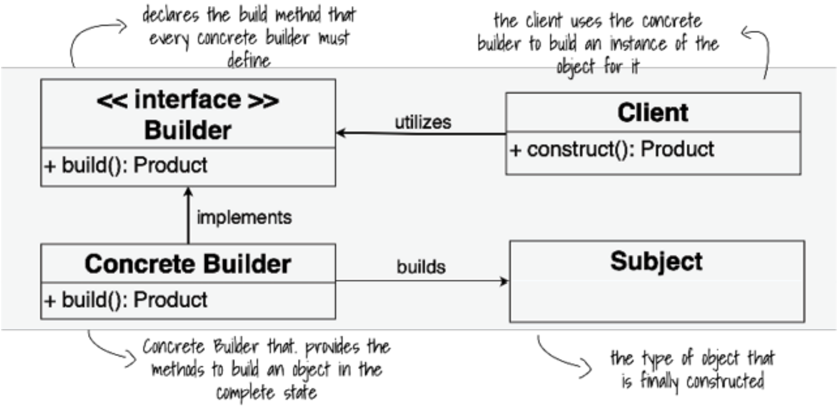

declares the build method that every concrete builder must define
the client uses the concrete builder to build an instance of the object for it
<< interface >>
Builder
+ build(): Product
utilizes
Client
+ construct(): Product
implements
Concrete Builder
+ build(): Product
builds
Subject
Concrete Builder that provides the methods to build an object in the complete state
the type of object that is finally constructed

***Figure 4.15**: Builder design pattern*

In the case of a large, complex object that has multiple dependencies and requires processing before they can be acquired, multiple builders can be created, each devoted to the task of building only one of the dependencies, and finally, one last builder can be delegated the task of invoking those builders to gather the dependencies and build the complex object.

The builder design pattern provides a generic approach to the process of building an object with different representational states. It effectively reduces direct interaction with the constructors of the object and improves readability by providing properly named, self-descriptive methods. It can also be utilized to perform validation before exposing the constructed object.

Advantages and drawbacks

The builder design pattern has many advantages over the approach of direct interaction with the constructors. Let us discuss the benefits followed by the drawbacks of it:

- It reduces the number of constructors in the subject class.
- It provides method calls to construct an object one step at a time, reducing the direct interaction with the constructors and improving readability.

- It guards against the creation of inconsistent state objects by using validations.
- It can be used to build immutable objects, considering the dependent objects are also immutable; for example, the string builder class in Java uses the builder design pattern.

The drawbacks of the builder design pattern are as follows:

- Most of the code is duplicated since each builder is an approximate replica of the subject class.
- Each representational state requires a separate builder.

Usage

The following list outlines some of the scenarios relevant to builder design patterns:

- When there are multiple possible representations of an object, for example, a report can have multiple datasets to display, but not every user is authorized to see it
- It can be used to incrementally build an object while respecting the validations put in place. For example, a restaurant menu builder can make use of date and time information to include or exclude certain dishes from the daily menu.

Conclusion

Builder design pattern is a widely used creational design pattern that can be utilized to build different representational states of an object. It provides an abstraction layer that hides the complexity of object construction from its users while simultaneously improving readability by exposing methods that build the object in parts, step-by-step. It ensures that the object that is finally instantiated is in a complete state.

For a large complex object with multiple dependencies, each dependency can be constructed using a different builder and later combined to form one complex object. The builder classes can be developed independently or as a static inner member of the subject class itself.

In the next chapter, we will continue exploring the creational design patterns and will learn about the object pool design pattern that helps us construct and reuse a pool of objects.

Questions

In this section, we will sharpen our brains by indulging in some fun brain activities. Let us get started.

Find the odd one out

1. Builder, concrete builder, concrete subject, subject
2. Build, multiple constructors, step-by-step, task delegation
3. Readable method calls, immutable object, complete state, code duplication
4. Builder design pattern:
   a. Reduces the number of constructors in the subject class
   b. Builds the subject step-by-step
   c. Instantiates an object in a complete state
   d. Can build immutable objects
5. Concrete builders:
   a. Duplicate a lot of code
   b. Must implement the builder interface
   c. Build multiple types of objects
   d. Utilize a stepwise approach to construct an object
6. Builder design pattern utilizes:
   a. Abstraction
   b. Composition
   c. Threading
   d. Inheritance

Select two correct answers

1. Builder design pattern:
   a. Duplicates code

    b. Use multiple constructors
    c. Can produce immutable objects
    d. Builds an incomplete object
2. Constructors:
    a. Can have multiple parameters
    b. Cannot be passed final references
    c. Construct an object
    d. Cannot be overridden
3. A builder:
    a. Can be a static inner-class
    b. Can have multiple build methods
    c. Can call other builders
    d. Can have multiple constructors
4. What goes together:
    a. Code duplication, incomplete state, step-by-step
    b. Static inner class, code duplication, multiple constructors
    c. Step-by-step, complete state, immutable object
    d. Single build method, step-by-step, multiple constructors
5. What goes together:
    a. Multiple build methods, code duplication
    b. Multiple constructors, not null arguments
    c. Step-by-step, complete state
    d. Null arguments, concrete builder

Answers

This section provides the answers to the questions given.

Find the odd one out

1. Concrete subject
2. Task delegation

3. Code duplication
4. Reduces the number of constructors in the subject class
5. Duplicate a lot of code
6. Threading

Select two correct answers

1. Using multiple constructor, can produce immutable objects
2. Can have multiple parameters, construct an object
3. Can be a static inner-class, can have multiple constructors
4. Step-by-step, complete state, immutable object and single build method, step-by-step, multiple constructors
5. Multiple constructors, not null arguments, and step-by-step, complete state

CHAPTER 5

Recycle and Reuse

Introduction

Applications often repeat operations and make use of distinct object types to perform them. Construction of such objects is often considered costly, either due to the number of resources they consume or because there is a limit to how many such objects an application can create. You can think of many such object types, for example, threads of database connection objects.

When such costly objects are involved, it is practical to look for ways to reuse them rather than construct a new object whenever required. In the next section, we will discuss object reusability and maintainability more.

Structure

This chapter covers the following topics:

- Reusable objects
- Using an object pool
- Varying implementations
- The object pool design pattern
- Object pool implementation
- Advantages and drawbacks

- Usage

Objectives

By the end of this chapter, the readers will understand the benefits of reusing objects and learn about object pools that act as object containers. They will learn about the object pool design pattern and how it can be utilized to manage objects and improve application performance.

The chapter will also walk them through the implementation details of the object pool design pattern, some of its variations, the benefits it provides, and a few drawbacks, along with some of the places where this design pattern can be used.

Reusable objects

Object reusability is one of the important aspects of software design; it not only advocates the reuse of already defined libraries but also promotes the use of existing object instances to perform recurring operations. Applications today perform millions of operations and often do so in parallel. If we must reuse our object instances, a single instance might not be enough to support parallel operations. One of the preferred approaches is to have a pool of objects to support such a magnitude of operations.

The reusability of objects is favorable under certain scenarios:

- **Cost of object construction**: When the object consumes a lot of computing resources, for example, CPU cycles or memory, that could have an impact on application performance.
- **Object construction time**: When an object requires a large time to construct, possibly due to some I/O operations, that could account for a delay in processing operations.
- **Parallel processing**: When the application requires the ability to support parallel processing.
- **Limited memory**: When the application's memory allocation allows the construction of only a limited number of objects.

There could be additional scenarios apart from the ones mentioned above, those that are usually relevant to some specific programming tasks or business requirements.

The object pool design pattern

A pool of objects shown in *Figure 5.1* is essentially a group of pre-constructed object instances that can be acquired from the pool to perform some operation and then released to be reused again:

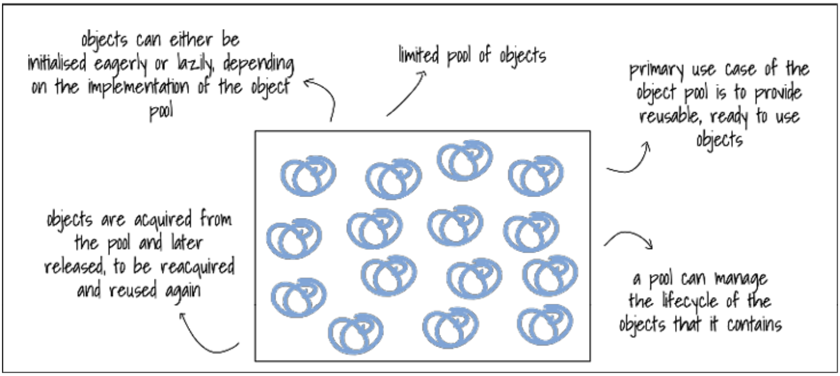

objects can either be initialised eagerly or lazily, depending on the implementation of the object pool
limited pool of objects
primary use case of the object pool is to provide reusable, ready to use objects
objects are acquired from the pool and later released, to be reacquired and reused again
a pool can manage the lifecycle of the objects that it contains

***Figure 5.1**: Pool of objects*

The **object pool design pattern** is a creational design pattern that suggests a way to reuse existing object instances rather than creating new ones on demand. It encourages using existing objects to perform computation and limit resource consumption.

The pool contains a finite number of object instances. The objects that form the pool can be constructed either eagerly or lazily, depending on the implementation of the pool. It contains similar objects; subtypes are also allowed depending on the pool implementation. *Figure 5.2* exhibits the architecture of the object pool design pattern:

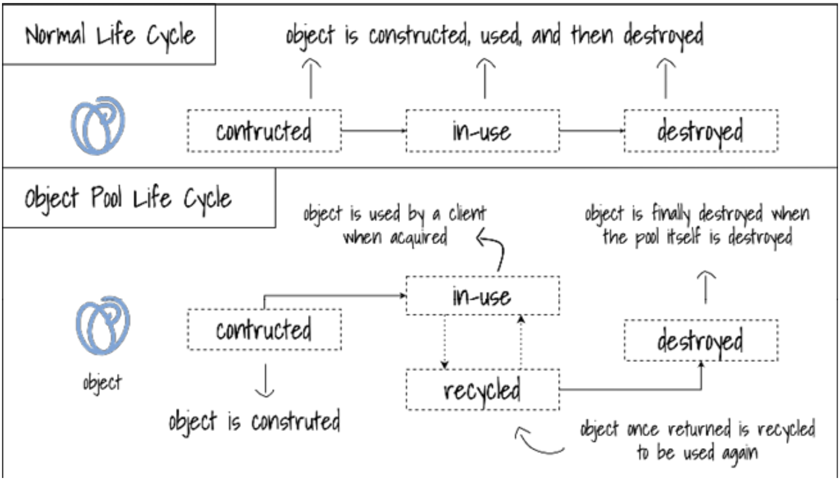

Normal Life Cycle
object is constructed, used, and then destroyed
contructed
in-use
destroyed
Object Pool Life Cycle
object is used by a client when acquired
object is finally destroyed when the pool itself is destroyed
in-use
contructed
destroyed
object
recycled
object is construted
object once returned is recycled to be used again

Figure 5.2: *Object pool design pattern*

The objects in the pool can be initialized well in advance so that they can be used to perform the intended functionality without creating new object instances. The objects from the pool can be acquired, used, and then returned to the pool to be reused again, either by the same or another client simultaneously accessing the object pool.

The object once acquired is no longer available in the pool until it is returned to it. For this reason, the object must be immediately returned to the pool after its use. This ensures that the objects in the pool can be made available to other clients when required, with as little delay as possible. The client must ensure that the acquired objects are returned to the pool and not destroyed.

The lifecycle of the objects that form the pool is different from that of all other objects of the application that are not part of the pool. An object's lifecycle usually contains stages, which include **constructed**, **in-use**, and **destroyed**, in the sequence mentioned. For objects that are part of the pool, this cycle is a bit circular, that is, **constructed**, **in-use**, **recycled**, and **destroyed**. This increased complexity requires additional effort in implementing an object pool. *Figure 5.3* exhibits the lifecycle of the pool objects:

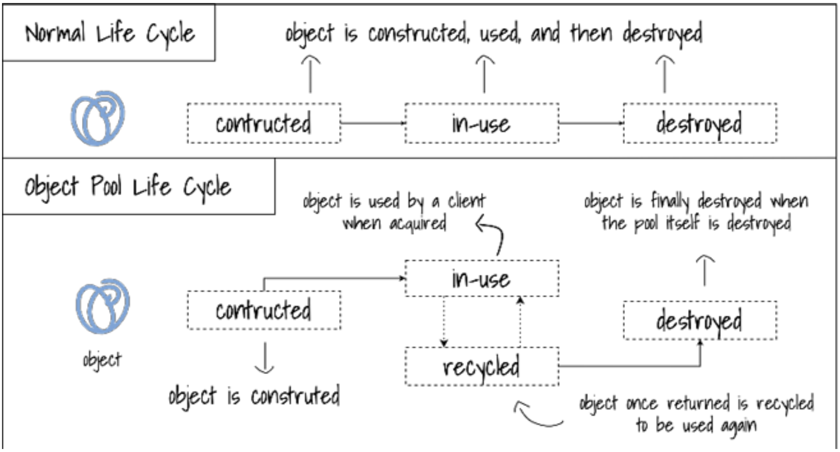

Normal Life Cycle
object is constructed, used, and then destroyed
contructed
in-use
destroyed
Object Pool Life Cycle
object is used by a client when acquired
object is finally destroyed when the pool itself is destroyed
in-use
contructed
destroyed
object
recycled
object is construted
object once returned is recycled to be used again

***Figure 5.3**: Lifecycle of pool objects*

Initializing pool objects at the time of pool construction ensures that these objects can be used straight away once the application is up and running. Otherwise, this could take up time to initialize pool objects, incurring delays before the objects can be used.

An object pool can manage objects in various ways. It may choose to initialize a few pool objects in advance and wait for them to be acquired before initializing new ones in the pool. It could even release some of the long-waiting, not-in-use objects to memory and later initialize new objects when required. The object pool design pattern suggests a way to form a pool of objects, but it is not limited to a single implementation.

As shown in *Figure 5.4*, objects in a pool are of a similar kind, reusable, have a distinct lifecycle, and have a higher cost of construction. The pool itself can have varying implementations and can support request queueing and lazy initialization of pool objects. Some of the characteristics mentioned above are definitive, while others depend on the implementation.

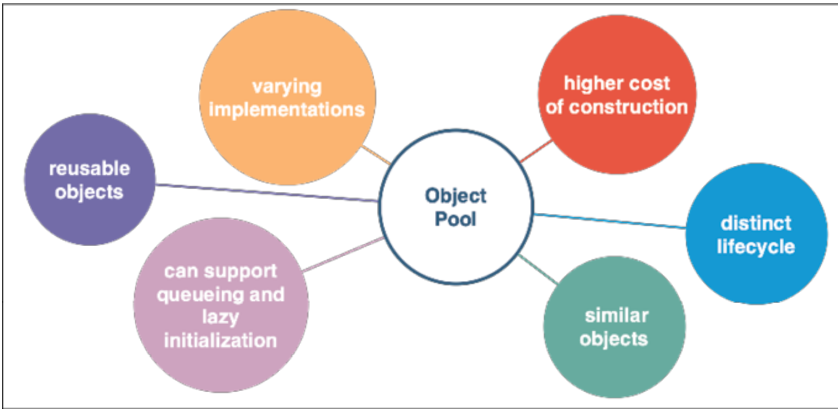

varying implementations
higher cost of construction
reusable objects
Object Pool
distinct lifecycle
can support queueing and lazy initialization
similar objects

***Figure 5.4**: Characteristics of object pool*

Using an object pool

A pool of objects is desirable when there is a requirement to work with a large set of objects that are expensive to initialize, consume high resources, and have repeated usage. In such scenarios, an object pool design is highly suitable as it provides pre-initialized, ready-to-use objects that can be acquired, used, and later released to be reused by other clients.

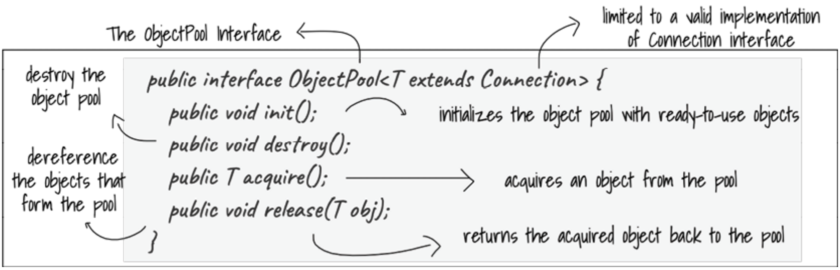
An object pool acts as a container for pre-initialized objects. When there is a requirement for an object, it can be acquired from the pool instead of creating a new one. When requested, the pool will return an available object that is effectively recycled. If the pool reaches its limit, it could either queue the incoming requests and put the requesters on wait until an object is available to be acquired, or it can throw an exception. Once the acquired object is no longer needed, it must be returned to the pool, where it will be recycled to prepare it for reuse. The recycling process must invalidate all the existing references of the returned object so that it is no longer accessible to its previous acquirer. The recycling process must also reset the object's state, effectively making it a new object.

If the object's state is not reset, it could exhibit wrong behavior. If the object is responsible for computations, it could output invalid results. If the object safeguards access to data, it could allow access based on its previous state, resulting in data theft. If the object provides access to a resource, for example, a database, it could end up providing unauthorized access. There could be other scenarios where a situation could go wrong if the object's state is not managed properly, and therefore, it is of utmost importance to ensure that the object is properly recycled before reuse.

Object pool implementation

An object pool has three main components. The first one is the interface that defines the contract, the second one is the class implementation that abides by it, and the third one is the object that is part of the pool.

In *Figure 5.5*, the interface takes advantage of Java's generic types and restricts the pool to contain only a specific type of object; subtypes are allowed. The ability to define the type of object that can be used within the implementing class allows for the reuse of the interface.



```
public interface ObjectPool<T extends Connection> {
    public void init();
    public void destroy();
    public T acquire();
    public void release(T obj);
}
```

The ObjectPool Interface
limited to a valid implementation of Connection interface
destroy the object pool
initializes the object pool with ready-to-use objects
dereference the objects that form the pool
acquires an object from the pool
returns the acquired object back to the pool

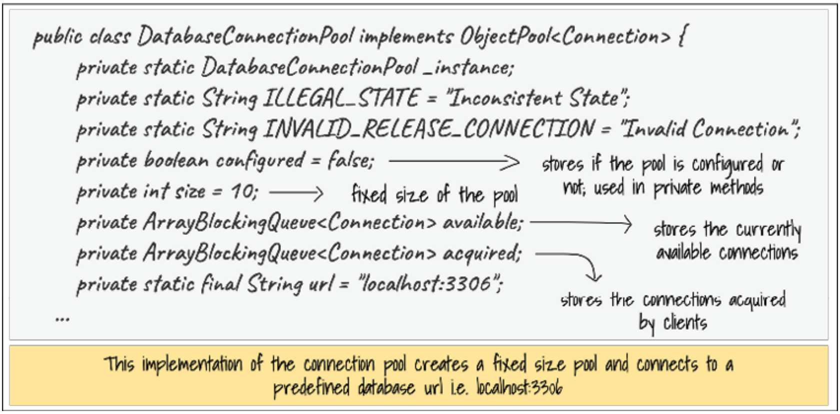
***Figure 5.5**: The object pool interface*

The interface has the following four methods:

- **init**: Constructs and initializes the object pool with ready-to-use, reusable objects that can be acquired, used, and then returned when no longer required.
- **destroy**: Destroys the object pool and de-references the objects that form the pool, allowing them to be garbage collected.
- **acquire**: It is used to acquire an object from the pool.
- **release**: It is used to return an object to the pool.

Now that we have examined the design of the object pool design pattern, including the interface that the concrete classes must implement and the methods exposed by that interface. We have also discussed when an object pool should be constructed and utilized, and some of the smaller details that the developer should work on while implementing an object pool. It is time to implement an object pool and discuss the different implementation approaches.

An object pool can be implemented in many ways depending on the use case and the frequency with which the objects are acquired from the pool. Objects that form the pool can be instantiated at the same time when the pool is constructed, or the pool could be partially filled with the objects at the time of its construction, instantiating only a few of them initially and deferring the instantiation of the rest of the objects when required. The remaining objects can be instantiated one at a time when there is a request to acquire an object from the pool, and all the existing pool objects are already acquired, but the pool size is still not reached. We will discuss these and a few other variations on the object pool implementation in the following sections of the chapter. For now, let us move ahead and implement our first object pool. *Figure 5.6* exhibits the implementation of a pool of database connection objects:



```
public class DatabaseConnectionPool implements ObjectPool<Connection> {
    private static DatabaseConnectionPool _instance;
    private static String ILLEGAL_STATE = "Inconsistent State";
    private static String INVALID_RELEASE_CONNECTION = "Invalid Connection";
    private boolean configured = false;
    private int size = 10;
    private ArrayBlockingQueue<Connection> available;
    private ArrayBlockingQueue<Connection> acquired;
    private static final String url = "localhost:3306";
  ...
```

stores if the pool is configured or not; used in private methods

fixed size of the pool

stores the currently available connections

stores the connections acquired by clients

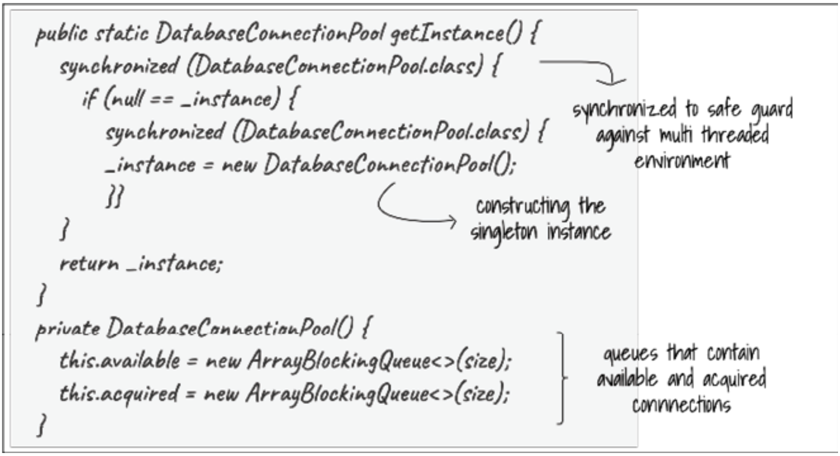
This implementation of the connection pool creates a fixed size pool and connects to a predefined database url i.e. localhost:3306

***Figure 5.6**: Fixed-size connection pool*

In the preceding example, we have used the approach of eager initialization, that is, instantiating all the objects in the pool at the time of the pool construction itself. We will use Java's standard connection interface for compatibility with any Java implementations. We will also use some basic multi-threading to safeguard our pool objects from concurrent access. And finally, for storing the objects in the pool, we will use the standard queue implementations.

Figure 5.6 exhibits various data members of our object pool implementation. Our implementation uses two blocking queues, one each for **available** and **acquired** connection objects, a variable to store the size of the pool, and a Boolean variable that stores the status of the pool, that is, whether it is configured or not. Once the connection pool is initialized with the connection objects, the pool's status is changed to configured.

Using two queues in the pool helps with managing the acquired and available objects. A single queue is difficult to manage, as an already acquired object could be reused even before it is released by the previous owner. *Figure 5.7* exhibits an implementation that constructs a fixed-size pool:



```
public static DatabaseConnectionPool getInstance() {
   synchronized (DatabaseConnectionPool.class) {
      if (null == _instance) {
         synchronized (DatabaseConnectionPool.class) {
         _instance = new DatabaseConnectionPool();
         }}
   }
   return _instance;
}
private DatabaseConnectionPool() {
   this.available = new ArrayBlockingQueue<>(size);
   this.acquired = new ArrayBlockingQueue<>(size);
}
```

synchronized to safe guard against multi threaded environment

constructing the singleton instance

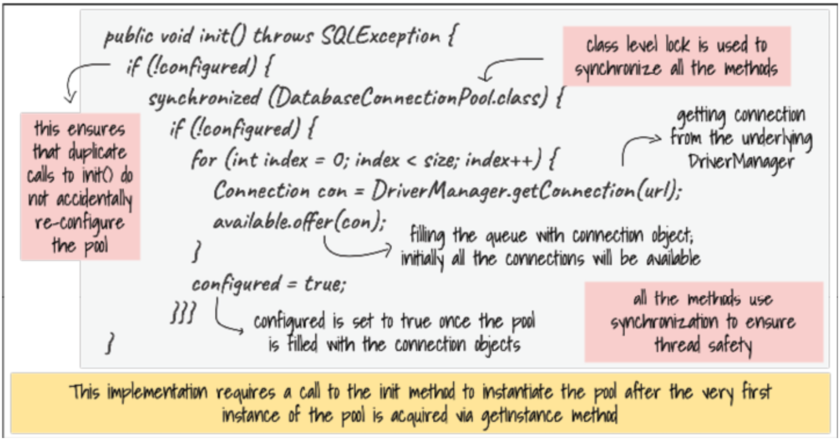
queues that contain available and acquired connections

***Figure 5.7**: Connection pool constructor*

It is the simplest of the approaches that allows us to easily understand the object pool and its design pattern. It uses the singleton design pattern, as discussed in *Chapter 1*, *Enlighten Yourself*, to ensure that there is only a single instance of the connection pool available at any point in time. Using a singleton ensures that only the desired number of connections are open, leaving no space for error. Having a single pool of database connections also makes it easier to manage and use the connections.

Pool initialization

Initializing the object pool is the first operation that is performed on the pool. This operation fills the pool with the desired number of object instances. In *Figure 5.8*, the **init** method initializes the pool with a fixed number of connection objects:



```
public void init() throws SQLException {
  if (!configured) {
    synchronized (DatabaseConnectionPool.class) {
      if (!configured) {
        for (int index = 0; index < size; index++) {
          Connection con = DriverManager.getConnection(url);
          available.offer(con);
        }
        configured = true;
      }}}
}
```

class level lock is used to synchronize all the methods

this ensures that duplicate calls to init() do not accidentally re-configure the pool

getting connection from the underlying DriverManager

filling the queue with connection object; initially all the connections will be available

configured is set to true once the pool is filled with the connection objects

all the methods use synchronization to ensure thread safety

This implementation requires a call to the init method to instantiate the pool after the very first instance of the pool is acquired via getInstance method

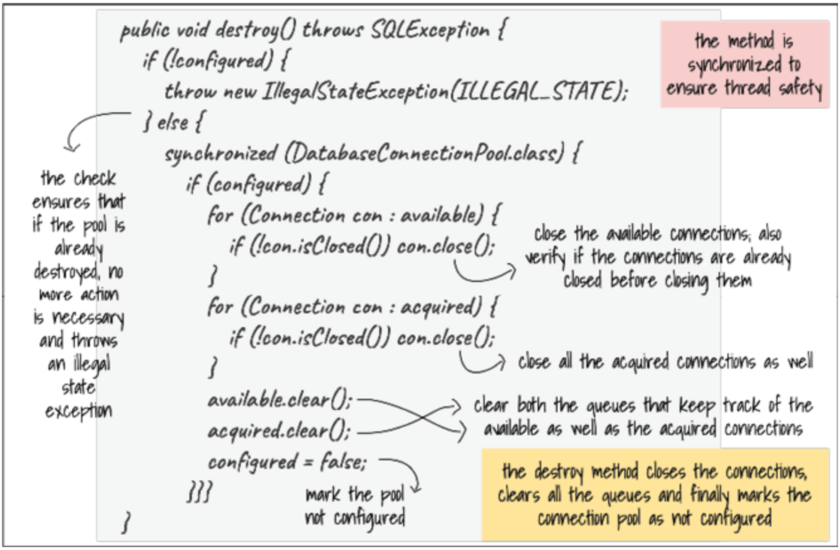
***Figure 5.8**: Initialize the object pool*

In order to ensure that the pool does not get initialized more than once or an already initialized pool is not overridden with new database connections, the **double-check locking** technique is used, which safeguards the block from being accessed by more than one thread simultaneously. The threads are only allowed to initialize the pool if the value **configured** is **false**. It is set to **true** once the pool is initialized and before the thread leaves the synchronized block.

Pool destruction

Destroying the object pool is the last operation that should be performed on the pool. This operation releases all the objects in the pool and sets its state to **not configured**.

In *Figure 5.9*, the **destroy** method closes all the database connections that are either available or acquired, clears both the queues, and sets the value of the configured variable to **false**:



```
public void destroy() throws SQLException {
  if (!configured) {
    throw new IllegalStateException(ILLEGAL_STATE);
  } else {
    synchronized (DatabaseConnectionPool.class) {
      if (configured) {
        for (Connection con : available) {
          if (!con.isClosed()) con.close();
        }
        for (Connection con : acquired) {
          if (!con.isClosed()) con.close();
        }
        available.clear();
        acquired.clear();
        configured = false;
      }}}
}
```

the method is synchronized to ensure thread safety

the check ensures that if the pool is already destroyed, no more action is necessary and throws an illegal state exception

close the available connections; also verify if the connections are already closed before closing them

close all the acquired connections as well

clear both the queues that keep track of the available as well as the acquired connections

mark the pool not configured

the destroy method closes the connections, clears all the queues and finally marks the connection pool as not configured

***Figure 5.9**: Destroy the pool*

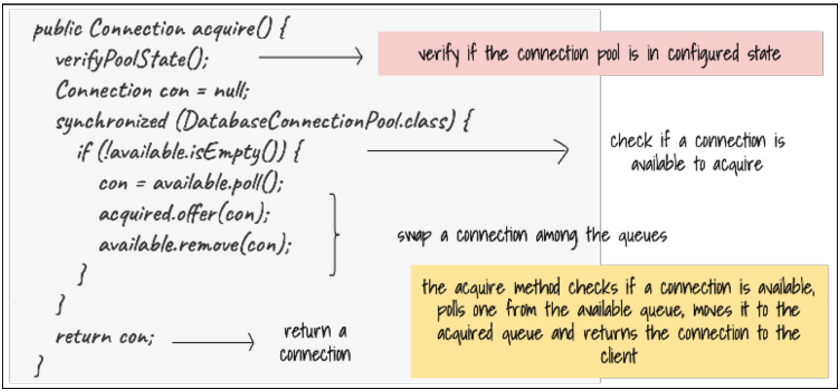
If the destroy method is called and the pool has already been destroyed, the request throws an exception. Our implementation does not wait for the acquired connections to be released. This is deliberately done to keep the example simple and to focus on the design pattern.

The object pool design pattern provides a framework to create and manage a pool of objects, and there are many ways to implement it. As discussed earlier, a pool of objects could be initialized eagerly or lazily, it could support many different types of objects, and could have many different, if not unique, ways to acquire an object and release it back to the pool. The implementations depend solely on the requirement, the design of the pool, and the type of data structures used to support the objects that form the pool.

Each connection object may have a reference to its current owner, which could help with validating the release process to ensure the connection is released by the correct owner, since the reference to an already released connection could be held up by some previous owner, and an accidental request could be triggered to release it.

Acquiring objects

Acquiring an object from the pool is a straightforward process. The mandatory requirement for it to succeed is to have an object available in the pool. *Figure 5.10* exhibits a simple algorithm that allows for the acquisition of an object from the object pool. In this implementation, the object is moved from the **available** queue to the **acquired** queue to mark it unavailable until it is released.

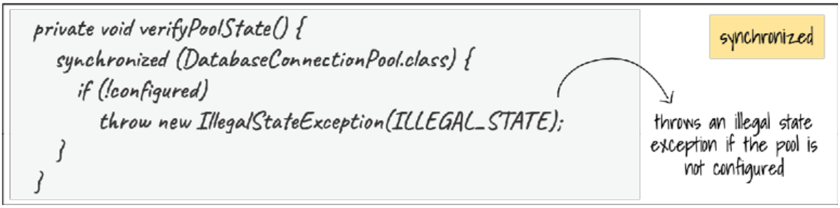


```
public Connection acquire() {
    verifyPoolState();
    Connection con = null;
    synchronized (DatabaseConnectionPool.class) {
        if (!available.isEmpty()) {
            con = available.poll();
            acquired.offer(con);
            available.remove(con);
        }
    }
    return con;
}
```

verify if the connection pool is in configured state
check if a connection is available to acquire
swap a connection among the queues
return a connection
the acquire method checks if a connection is available, polls one from the available queue, moves it to the acquired queue and returns the connection to the client

***Figure 5.10**: Acquire objects*

Before fulfilling a request to acquire an object, the pool is verified to be in the correct state, as exhibited in *Figure 5.11*. The pool is in the correct state if it has been properly initialized and has not been destroyed; otherwise, an exception is thrown.



```
private void verifyPoolState() {
    synchronized (DatabaseConnectionPool.class) {
        if (!configured)
            throw new IllegalStateException(ILLEGAL_STATE);
    }
}
```

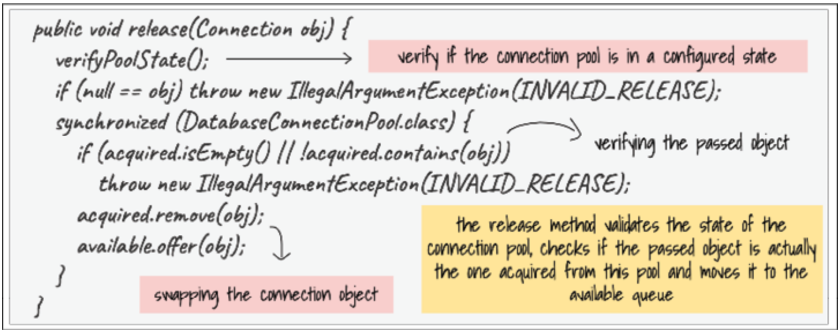
synchronized
throws an illegal state exception if the pool is not configured

***Figure 5.11**: Verify pool state*

This ensures that the request to acquire a connection does not lead to any surprises and helps in the graceful handling of such requests.

Releasing objects

To release an object back to the pool, the owner must call the release method and pass a valid connection object. *Figure 5.12* demonstrates a simple algorithm that allows the release of an object back to the object pool. We must ensure that the passed object was indeed acquired from the pool; otherwise, the release method will throw an exception.



```
public void release(Connection obj) {
  verifyPoolState();
  if (null == obj) throw new IllegalArgumentException(INVALID_RELEASE);
  synchronized (DatabaseConnectionPool.class) {
    if (acquired.isEmpty() || !acquired.contains(obj))
      throw new IllegalArgumentException(INVALID_RELEASE);
    acquired.remove(obj);
    available.offer(obj);
  }
}
```

verify if the connection pool is in a configured state

verifying the passed object

swapping the connection object

the release method validates the state of the connection pool, checks if the passed object is actually the one acquired from this pool and moves it to the available queue

***Figure 5.12**: Release objects*

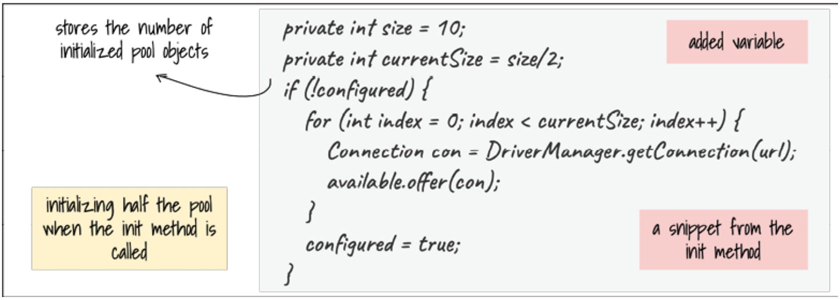
When an already released object is passed to the release method, an exception is thrown as well. In this implementation, the released object is moved from the **acquired** queue to the **available** queue to make it available for reuse. It should be noted that this implementation does not provide an out-of-the-box solution to ensure that the acquired objects are indeed released back to the pool, and therefore, the acquirer must ensure their release after use.

Varying implementations

An object pool can be implemented in many ways, depending on the requirements and the choice of data structures. In this section, we will explore some of those variations, for example, partial initialization or lazily created pool objects. We will also discuss improving our pool design to create a more robust object pool.

Partial initialization

An object pool can be filled with a small number of objects at the time of pool construction while instantiating more objects on demand. It is called the **lazy initialization process,** where only a few objects take up space when the pool is first constructed. *Figure 5.13* exhibits the changes we have made to our earlier approach to constructing and initializing an object pool:

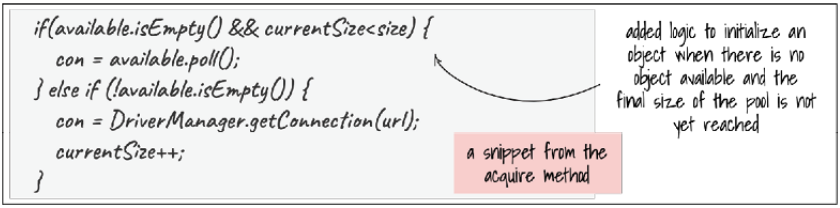


```
private int size = 10;
private int currentSize = size/2;
if (!configured) {
  for (int index = 0; index < currentSize; index++) {
    Connection con = DriverManager.getConnection(url);
    available.offer(con);
  }
  configured = true;
}
```

stores the number of initialized pool objects
added variable
initializing half the pool when the init method is called
a snippet from the init method

***Figure 5.13**: Partial initialization I*

The new implementation only constructs half the objects and adds them to the pool. The rest of the objects are constructed over time when the need arises. *Figure 5.14* exhibits the changes in the **acquire** method:



```
if(available.isEmpty() && currentSize<size) {
  con = available.poll();
} else if (!available.isEmpty()) {
  con = DriverManager.getConnection(url);
  currentSize++;
}
```

added logic to initialize an object when there is no object available and the final size of the pool is not yet reached
a snippet from the acquire method

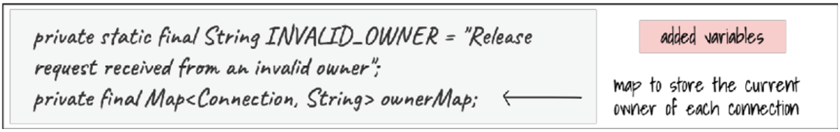
***Figure 5.14**: Partial initialization II*

The **acquire** method can now construct and add a new object to the pool if there are no objects available to be acquired and the pool size is not reached.

Object release

Releasing an object back to the object pool is an important task and should be handled with utmost care. When an object is acquired from the pool, the acquirer is termed the owner. If the acquired object is mistakenly released by some other thread and not the owner, this could have consequences, as the object might already have been reacquired by some other thread, while the earlier owner still has its reference.

A simple solution to this problem is to keep a map of all the connections and their present owner threads. The map is updated each time a connection object is acquired from or released back to the pool. The same map is used to verify whether the request to release a connection object back to the pool is made by the current owner of that connection. In case of an invalid request, an exception is thrown to safeguard the consistency of the pool. *Figure 5.15* exhibits this approach, where we have introduced a map of connection objects and a corresponding string value to store the name of the thread that currently owns that connection:



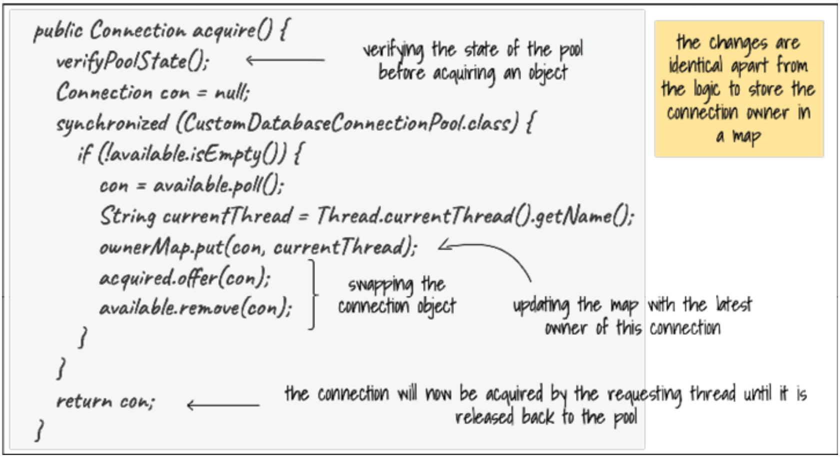
```
private static final String INVALID_OWNER = "Release
request received from an invalid owner";
private final Map<Connection, String> ownerMap;
```

added variables

map to store the current owner of each connection

Figure 5.15: *Object release I*

Figure 5.16 demonstrates the changes required in the **acquire** method for this approach to work:



```
public Connection acquire() {
   verifyPoolState();
   Connection con = null;
   synchronized (CustomDatabaseConnectionPool.class) {
      if (!available.isEmpty()) {
         con = available.poll();
         String currentThread = Thread.currentThread().getName();
         ownerMap.put(con, currentThread);
         acquired.offer(con);
         available.remove(con);
      }
   }
   return con;
}
```

verifying the state of the pool before acquiring an object

the changes are identical apart from the logic to store the connection owner in a map

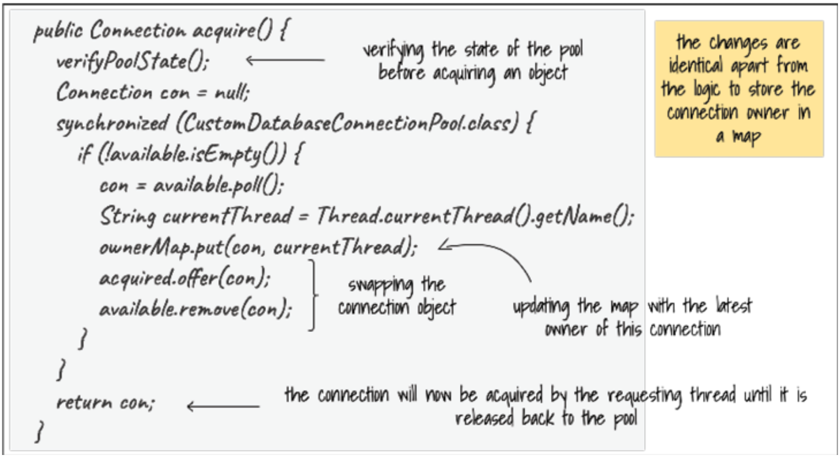
updating the map with the latest owner of this connection

swapping the connection object

the connection will now be acquired by the requesting thread until it is released back to the pool

Figure 5.16: *Object release II*

As demonstrated in *Figure 5.17*, every time a connection is acquired or released, the map is also updated. When a connection is acquired, the map is updated to store the owner's relationship of the connection with the thread that acquires it. When the object is released, the relationship is severed by associating the connection with a **null** value.



```
public Connection acquire() {
  verifyPoolState();
  Connection con = null;
  synchronized (CustomDatabaseConnectionPool.class) {
    if (!available.isEmpty()) {
      con = available.poll();
      String currentThread = Thread.currentThread().getName();
      ownerMap.put(con, currentThread);
      acquired.offer(con);
      available.remove(con);
    }
  }
  return con;
}
```

verifying the state of the pool before acquiring an object

the changes are identical apart from the logic to store the connection owner in a map

swapping the connection object

updating the map with the latest owner of this connection

the connection will now be acquired by the requesting thread until it is released back to the pool

***Figure 5.17**: Object release III*

Advantages and drawbacks

Let us look at the benefits and drawbacks of the object pool design pattern. First, we will discuss the benefits, followed by the drawbacks:

- It provides ready-to-use, reusable objects that reduce the need for on-demand creation of objects.
- It improves the performance of the application by reducing resource consumption.

The drawback of the object pool design pattern is that if the objects are not released properly, it could adversely impact the behavior and performance of the application.

Usage

The following list outlines some of the scenarios that are relevant to building an object pool:

- When the objects are expensive to construct and are supposed to be reused.
- When the application requires the ability to support parallel processing.

Conclusion

Object pool design pattern is a widely used creational design pattern that can be utilized to build a pool of reusable objects. It provides a framework to pre-construct a pool of similar types of objects that can be reused to improve performance. The design pattern helps to reduce the time required for constructing objects and facilitates loading objects at the time of system start-up.

An object pool can be used when the objects consume a lot of computing resources, or when the objects require a large amount of time to construct. An object pool is also useful in scenarios where the objects are candidates for parallel processing or the system has a limited memory.

In the next chapter, we will discuss the prototype design pattern and will learn about cloning existing objects to create new ones.

Questions

In this section, we will sharpen our brains by indulging in some fun brain activities.

Find the odd one out

1. High resource consumption, reusable, multi-threaded, bulk initialization.
2. Fix-sized, partially initialized, infinitely large, variable-sized.
3. Reusable object, immutable object, singleton object, mutable object.
4. Object pool design pattern:
    a. Helps to reduce time complexity
    b. Requires a large memory to implement
    c. Provides a framework to manage a pool of objects
    d. Requires parallel processing capabilities
5. Objects that form an object pool:
    a. Are of the same type
    b. Must implement the object pool interface

   - c. May consume high resources
   - d. Can be lazily initialized by the pool
6. Valid methods of the object pool interface are
   - a. Destruct
   - b. Init
   - c. Release
   - d. Acquire

Select two correct answers

1. Object pool design pattern:
   - a. Provides one object at a time
   - b. Can work in a multi-threaded environment
   - c. Is suitable for infinite objects
   - d. Can only be used for non-immutable objects
2. Methods supported by the object pool design pattern:
   - a. Construct
   - b. Acquire
   - c. Destruct
   - d. Destroy
3. Which objects would you make a pool of:
   - a. Immutable objects
   - b. Private objects
   - c. Resource-consuming objects
   - d. Singletons
4. What goes together:
   - a. Init, acquire, release, destroy
   - b. High resource consumption, object pool
   - c. Object pool, singletons, immutable objects
   - d. Destruct, get, put, initialize

Answers

This section provides the answers to the questions given.

Find the odd one out

1. High resource consumption
2. Infinitely large
3. Mutable object
4. Requires a large memory to implement
5. Are of the same type
6. destruct

Select two correct answers

1. Provides one object at a time, can work in a multi-threaded environment
2. Acquire, destroy
3. Immutable objects, resource-consuming objects
4. Init, acquire, release, destroy, and high resource consumption, object pool

CHAPTER 6

Prototypes

Introduction

Object creation is an essential part of software development. When building software, we often need to create multiple objects, *but what if those objects are complex, expensive to create, or need to be duplicated with slight modifications?* This is why the prototype design pattern is essential, as it can be utilized to recreate objects using a pre-initialized object known as a **prototype**. The prototype object acts as a blueprint and can be cloned to recreate an object with the exact same state as the prototype itself.

Structure

This chapter covers the following topics:

- Prototypes
- The prototype design pattern
- Implementing the prototype pattern
- Prototype registry
- Importance of a registry
- Summary
- Advantages and drawbacks
- Usage

Objectives

By the end of this chapter, the readers will learn about object prototypes and how they can be used to recreate complex objects with state spread across multiple layers. The chapter will also discuss the implementation details of the prototype design pattern, the benefits it provides, and a few drawbacks, along with some of the places where this design pattern can be used.

Prototypes

A prototype object is an existing object that is used as a *template* to create new objects by cloning or copying it, instead of instantiating a brand-new one from scratch. *Figure 6.1* exhibits a prototype object on the left and some cloned objects on the right:

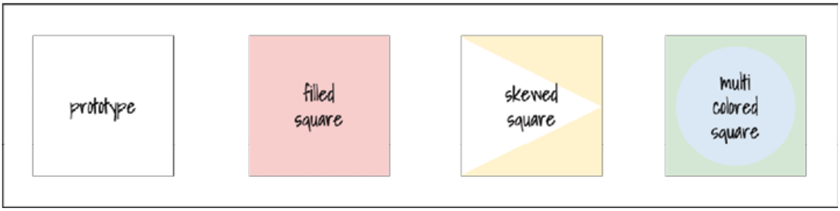


***Figure 6.1**: Prototype and clones*

A **prototype** allows a quick recreation of the base state of an object that otherwise could be a time-consuming and resource-intensive operation. This application of a prototype object makes it highly useful as it allows us to save a lot of time and resources.

In the **prototype design pattern**, the prototype object provides a way to duplicate itself, which is particularly useful when object creation is expensive (e.g., complex initialization, database fetch), you need many objects with similar configuration, or if you want to hide the complexity of object creation from the client.

The prototype design pattern

The prototype pattern is a creational design pattern that allows you to clone existing objects instead of creating new ones from scratch. Use of the prototype pattern is recommended when object creation is costly, and you want to duplicate an existing object with slight variations.

The prototype design pattern, as shown in the following figure, emphasizes reuse through cloning and avoids repeating complex setups by copying existing objects. It promotes decoupling from concrete classes. Clients do not need to know the exact class of the object they are creating, but rather just how to clone it.

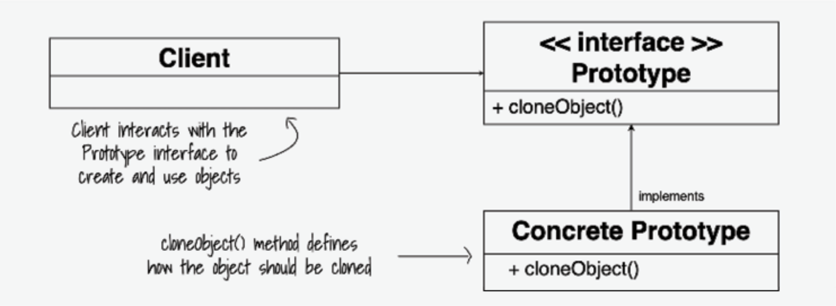

Client
<< interface >>
Prototype
+ cloneObject()
Client interacts with the Prototype interface to create and use objects
implements
Concrete Prototype
+ cloneObject()
cloneObject() method defines how the object should be cloned

***Figure 6.2**: Prototype design pattern*

The prototype design pattern relies on a clone method that copies the original object into a new one, as exhibited in *Figure 6.2*. This pattern has an interface called **prototype** that all the concrete prototypes must implement.

This pattern provides runtime flexibility that allows for the objects to be cloned and modified at runtime based on dynamic needs. It also encapsulates the construction logic and hides the complexity of object setup from the client.

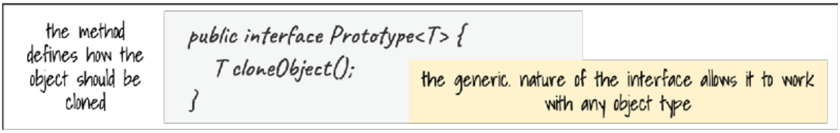
Implementing the prototype pattern

The example that is discussed throughout the chapter is of a word processing application that provides templates for creating documents. These templates or prototypes can be used to create a new document with the same characteristics as the prototype itself.

We will learn about the implementation of the prototype design pattern and how it can be used to recreate a certain state of an object with ease.

Prototype interface

The prototype interface, as exhibited in *Figure 6.3*, is at the core of this design pattern. It has a single method that allows for providing a definition to clone an object. The object in question here is a prototype, an existing object that the other objects are supposed to replicate.



```
public interface Prototype<T> {
    T cloneObject();
}
```

the method defines how the object should be cloned

the generic nature of the interface allows it to work with any object type

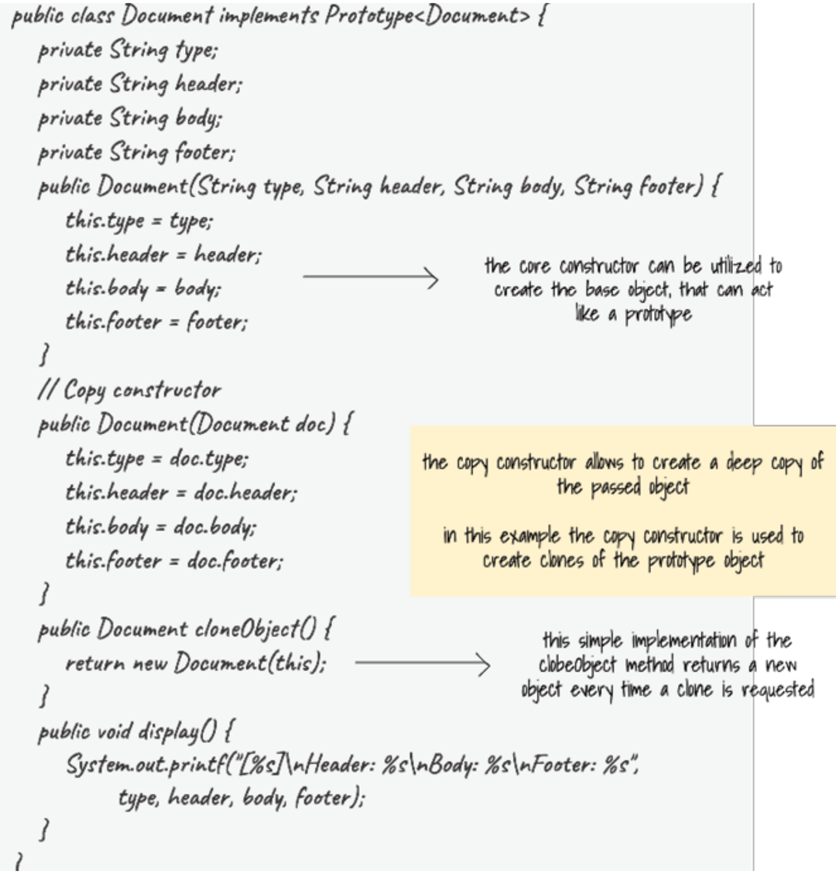
Figure 6.3: *Interface: Prototype*

The generic nature of the interface allows it to work with any type of object, and the method defines how the object should be cloned.

Prototype implementations

The concrete implementations of the prototype interface provide the cloning strategy and the definition of the prototype. The prototype allows us to reduce the time and resources required to construct the same object from scratch. It reflects the base state of an object, which can be cloned to create other objects that can be altered or built upon to reflect different states of it.

Figure 6.4 exhibits a concrete implementation of the prototype interface. It has a **cloneObject** method that uses the built-in copy constructor to clone an object:



```
public class Document implements Prototype<Document> {
    private String type;
    private String header;
    private String body;
    private String footer;
    public Document(String type, String header, String body, String footer) {
        this.type = type;
        this.header = header;
        this.body = body;
        this.footer = footer;
    }
    // Copy constructor
    public Document(Document doc) {
        this.type = doc.type;
        this.header = doc.header;
        this.body = doc.body;
        this.footer = doc.footer;
    }
    public Document cloneObject() {
        return new Document(this);
    }
    public void display() {
        System.out.printf("[%s]\nHeader: %s\nBody: %s\nFooter: %s",
                type, header, body, footer);
    }
}
```

the core constructor can be utilized to create the base object, that can act like a prototype

the copy constructor allows to create a deep copy of the passed object

in this example the copy constructor is used to create clones of the prototype object

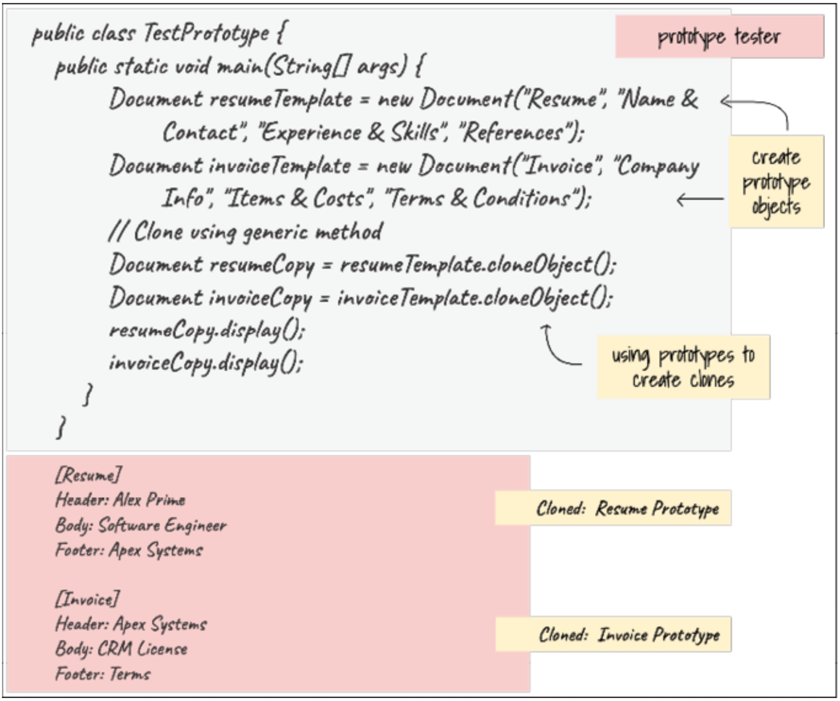
this simple implementation of the clobeObject method returns a new object every time a clone is requested

***Figure 6.4**: Prototype implementation*

The copy constructor is replicating the state of the current object instance. To create an instance in the first place, the default constructor should be used, and then the **cloneObject** method can be utilized to replicate it. This method allows us to create as many original prototypes as required and then use them as blueprints to create similar objects.

Testing the prototype

Figure 6.5 exhibits a code snippet that uses the prototype interface and its concrete implementation to test the prototype design pattern. It first creates two prototype objects of type **Document** and then creates replicas of them using the **cloneObject** method.



```
public class TestPrototype {
  public static void main(String[] args) {
        Document resumeTemplate = new Document("Resume", "Name &
              Contact", "Experience & Skills", "References");
        Document invoiceTemplate = new Document("Invoice", "Company
              Info", "Items & Costs", "Terms & Conditions");
        // Clone using generic method
        Document resumeCopy = resumeTemplate.cloneObject();
        Document invoiceCopy = invoiceTemplate.cloneObject();
        resumeCopy.display();
        invoiceCopy.display();
    }
  }
```

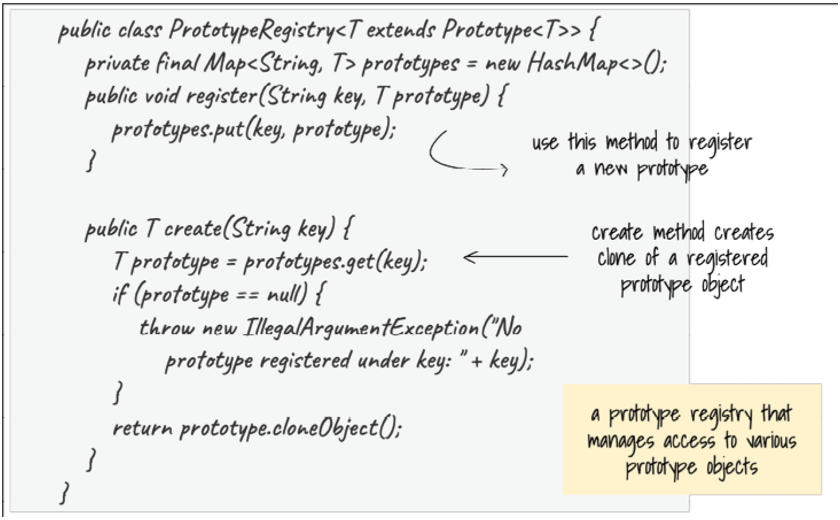
```
[Resume]
Header: Alex Prime
Body: Software Engineer
Footer: Apex Systems

[Invoice]
Header: Apex Systems
Body: CRM License
Footer: Terms
```

***Figure 6.5**: Test prototype*

The preceding figure also exhibits the initial state of the cloned objects. The prototypes are very useful when the requirement is to frequently create resource-heavy objects. One such object can be created as a prototype, and others can simply be a clone of it.

Prototype registry

In the prototype design pattern, the prototype registry is a central component that helps manage and access different prototype instances. *Figure 6.6* exhibits a prototype registry that allows for registering prototypes and storing them in a map. Once registered, a new object can be constructed from these prototypes using the create method.



```
public class PrototypeRegistry<T extends Prototype<T>> {
    private final Map<String, T> prototypes = new HashMap<>();
    public void register(String key, T prototype) {
        prototypes.put(key, prototype);
    }

    public T create(String key) {
        T prototype = prototypes.get(key);
        if (prototype == null) {
            throw new IllegalArgumentException("No
                prototype registered under key: " + key);
        }
        return prototype.cloneObject();
    }
}
```

use this method to register a new prototype

create method creates clone of a registered prototype object

a prototype registry that manages access to various prototype objects

***Figure 6.6**: Prototype registry*

A prototype registry acts like a catalog or repository of pre-configured objects that can be cloned on demand. Instead of creating new objects from scratch, the client fetches a prototype from the registry and clones it.

When using the prototype design pattern, the registry allows clients to access prototypes by key or identifier, clone them to create new instances, and avoid direct use of new or constructors.

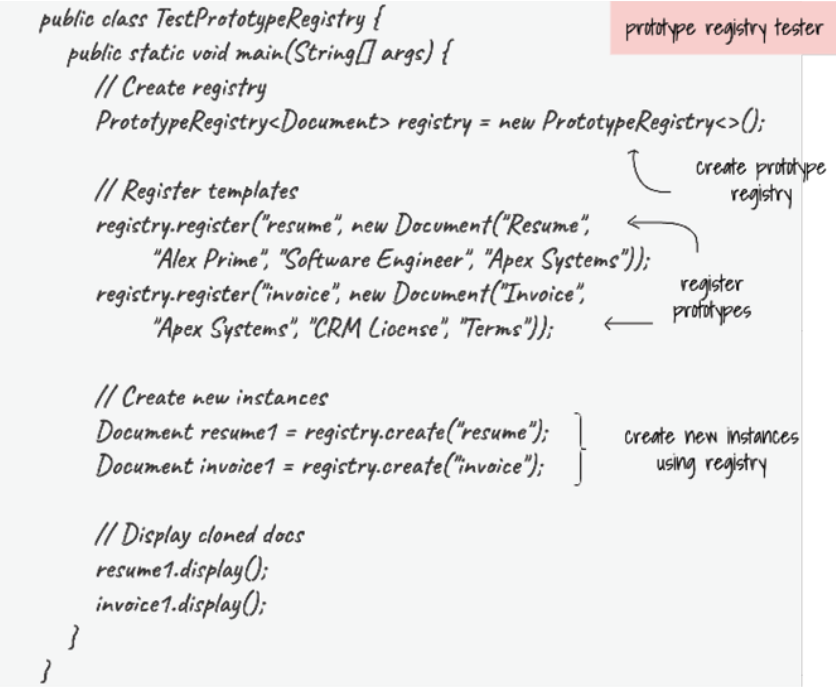
Importance of a registry

In the traditional prototype pattern, the client must hold and manage the prototype objects. As complexity grows, this becomes unmanageable.

The registry reduces the complexity by applying the following methods:

- **Centralizing object access**: Store all prototypes in one place.
- **Decoupling creation logic**: Clients do not need to know how to build the prototype.
- **Supporting runtime configuration**: You can register new prototypes at runtime.

Our prototype registry is utilized by a test class in *Figure 6.7* to manage and store prototype objects for creating copies of prototypes:



```
public class TestPrototypeRegistry {
    public static void main(String[] args) {
        // Create registry
        PrototypeRegistry<Document> registry = new PrototypeRegistry<>();

        // Register templates
        registry.register("resume", new Document("Resume",
                "Alex Prime", "Software Engineer", "Apex Systems"));
        registry.register("invoice", new Document("Invoice",
                "Apex Systems", "CRM License", "Terms"));

        // Create new instances
        Document resume1 = registry.create("resume");
        Document invoice1 = registry.create("invoice");

        // Display cloned docs
        resume1.display();
        invoice1.display();
    }
}
```

prototype registry tester
create prototype registry
register prototypes
create new instances using registry

Figure 6.7: *Test prototype registry*

Summary

The prototype design pattern is a creational design pattern that focuses on the concept of creating new objects by copying existing ones, known as **prototypes**, rather than instantiating them directly using constructors. This pattern is particularly useful when object creation is an expensive operation, either in terms of time, system resources, or complexity, such as when objects require deep configuration, involve database operations, or represent complex data structures. At its core, the pattern requires a base interface or abstract class that defines a **clone()** method, which each concrete class implements to return a copy of itself. This allows clients to duplicate objects without coupling to their specific classes, promoting flexibility and reducing code duplication. *Figure 6.8* exhibits a **Unified Modelling Language** (**UML**) diagram of the prototype design pattern updated with the class and interface types used in our example:

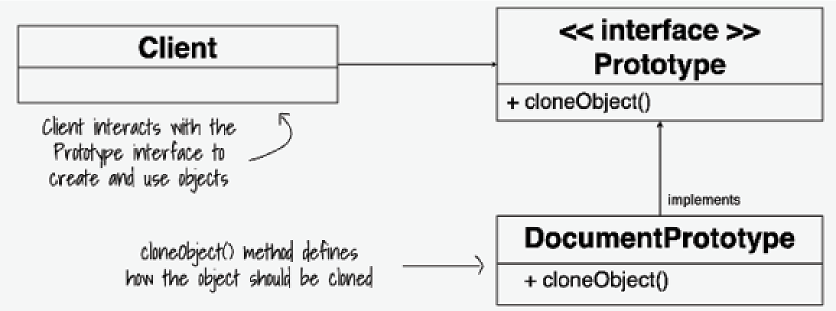

Client
<< interface >>
Prototype
+ cloneObject()
Client interacts with the Prototype interface to create and use objects
implements
DocumentPrototype
+ cloneObject()
cloneObject() method defines how the object should be cloned

Figure 6.8: *Prototype design pattern*

A key extension of this pattern is the prototype registry, a centralized structure that stores and manages a set of prototype instances. Clients can request a clone of any registered prototype by key, enabling dynamic and efficient creation of object variations at runtime. The prototype pattern is especially common in systems like graphical editors, game engines, and **user interface** (**UI**) frameworks, where numerous similar objects are created repeatedly. It supports both shallow and deep copying strategies, depending on the complexity and mutability of the object's internal state. While powerful, care must be taken to implement cloning correctly to avoid unexpected behavior, especially when dealing with nested or shared references. Overall, the prototype design pattern enhances performance, maintains cleaner code, and provides a flexible foundation for building scalable, object-rich systems.

Advantages and drawbacks

The following list outlines key benefits of the object pool design pattern:

- Avoids repetitive initialization logic.
- Improves performance via object reuse.
- Decouples client code from concrete class instantiation.

The drawbacks of the object pool design pattern are as follows:

- Improper cloning may lead to bugs, shared state issues, or memory leaks.

- Deep cloning can be costly in terms of memory and **central processing unit** (**CPU**), sometimes negating the performance benefits, particularly when objects have nested collections or large data structures.
- Cloned objects might carry hidden dependencies (e.g., configuration data, injected services), making the clone subtly different from a truly *fresh* instance and leading to hard-to-debug issues.

Usage

The following list outlines some of the scenarios that are relevant to using prototypes:

- Duplicating UI components (buttons, modals) with a common structure.
- Cloning characters in games with the same base attributes.
- Copying document or report templates with shared headers, styles, and footers.
- Creating preconfigured server or **virtual machine** (**VM**) instances in cloud orchestration.

Conclusion

The prototype design pattern offers a powerful and flexible approach to object creation, especially in scenarios where performance, configurability, or complexity are concerns. By enabling the cloning of pre-configured objects, it eliminates the need for repetitive instantiation logic and tightly coupled constructors. The addition of a prototype registry further streamlines object management, making it easy to maintain and extend reusable blueprints across an application. While it requires careful implementation, particularly with deep copies and mutable fields, the benefits in scalability, maintainability, and runtime efficiency make it a valuable tool in any developer's design pattern toolkit. When used appropriately, the prototype pattern can lead to cleaner, faster, and more adaptable code.

In the next chapter, we will discuss the adapter design pattern and will learn how to connect two incompatible interfaces to work with each other.

Questions

In this section, we will sharpen our brains by indulging in some fun brain activities. Let us get started.

Find the odd one out

1. Which of the following is not true for protypes
    a. Blueprints
    b. Reusable
    c. Not reusable
    d. Templates
2. Which of the following is not directly related to the prototype design pattern?
    a. Cloning
    b. Registry
    c. Constructor injection
    d. Deep copying
3. Which of the following is not a valid use case for the prototype pattern?
    a. Creating copies of graphic objects in a drawing tool
    b. Generating reports with varying templates
    c. Managing database connection pooling
    d. Creating enemy units in a video game
4. Which of the following is not a drawback of the prototype pattern?
    a. Complex cloning logic
    b. Memory overhead from storing prototypes
    c. Tight coupling to concrete classes
    d. Risk of shallow copying errors

Select two correct answers

1. Which two are key benefits of using the prototype design pattern?
    a. Reduces object creation time
    b. Prevents inheritance altogether
    c. Allows runtime configuration of objects
    d. Enforces a strict object lifecycle
2. Which two features are commonly associated with a prototype registry?
    a. Stores preconfigured object instances
    b. Automatically updates all clones when a prototype changes
    c. Provides access to cloning via identifiers
    d. Forces objects to be immutable

Answers

This section provides the answers to the questions given.

Find the odd one out

1. Not reusable
2. Constructor injection
3. Managing database connection pooling
4. Tight coupling to concrete classes

Select two correct answers

1. Reduces object creation time, allows runtime configuration of objects.
2. Stores preconfigured object instances, provides access to cloning via identifiers.

CHAPTER 7

Adapter

Introduction

When two or more systems interact with each other, it is not unusual to have similar operations represented in different forms and shapes. This is especially true when the systems are developed separately and have no prior knowledge of each other until they are integrated. The integration requires the two systems to interact, but before anything else, the systems need a way to understand each other.

The integration process must ensure that it restricts the changes required for integrating the systems to as few as possible; otherwise, we will end up modifying a lot more than required. In an ideal scenario, there should be no change required in the systems that are candidates for integration.

Structure

This chapter covers the following topics:

- Incompatible interfaces
- Adapter design pattern
- Building an adapter
- Summary

- Advantages and drawbacks
- Usage

Objectives

By the end of this chapter, the readers will learn how to integrate incompatible interfaces. Interfaces may not always support effortless integration; in fact, the developer might not even be aware of any possible integration requirement at the time of interface creation. Interfaces that express different functionalities are therefore hard to integrate without making any changes. They will also learn about the adapter design pattern and how to utilize it to integrate and reuse existing interfaces.

Finally, the chapter will walk them through the implementation details of the adapter design pattern, its benefits, and a few drawbacks, along with some of the places where this design pattern can be used.

Incompatible interfaces

You must have heard of this famous phrase—*no need to reinvent the wheel*; the cost of rewriting application code that already exists could be more than the actual estimations. Existing modules of code that have passed the test of time are usually efficient and do not require a rewrite, although some improvements could be on the cards.

The world of software has grown exponentially; they perform a variety of tasks and is often housed with algorithms that can be utilized in multiple business use cases. These common code blocks are usually written keeping in mind the reusability principle.

The crux here is that it is not always straightforward to integrate; what this means is that the interfaces do not always match, or the methods hosted by these common code blocks have different signatures. So, *does this mean we have to rewrite?* No, we do not. Instead, we can write adapter classes that help us integrate incompatible interfaces without modifying them.

Adapter design pattern

The adapter design pattern integrates two incompatible interfaces. It allows an interface implementation to mimic another interface and ensures no modifications in either of the existing interfaces. The adapter pattern provides a neat way to integrate already designed interfaces with no strings attached; the adapter implementation is a standalone class that extends no obligation to the existing code, something that can be removed if required without affecting the system as such.

In *Figure 7.1*, the client expects an interface of type *A*, but the one exposed by the existing system is of type *B*:

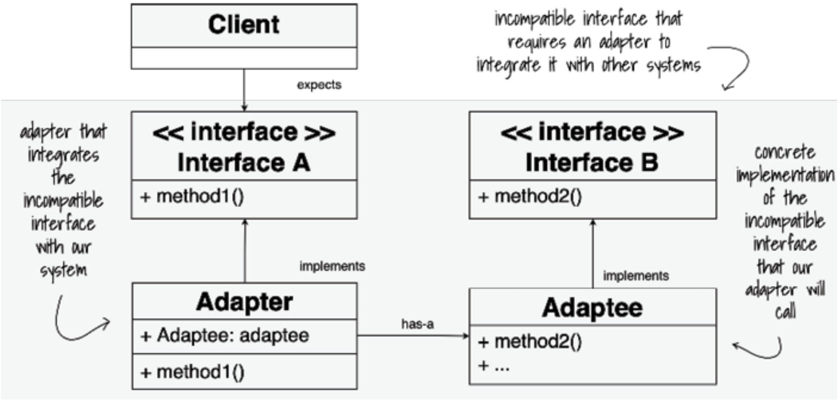

Client
expects
incompatible interface that requires an adapter to integrate it with other systems
adapter that integrates the incompatible interface with our system
<< interface >>
Interface A
+ method1()
<< interface >>
Interface B
+ method2()
concrete implementation of the incompatible interface that our adapter will call
implements
implements
Adapter
+ Adaptee: adaptee
+ method1()
has-a
Adaptee
+ method2()
+ ...

***Figure 7.1**: Adapter design pattern*

Although the system provides the required features, the incompatibility is restricting its usage. To overcome this restriction, one can make use of the adapter design pattern. Here is how it works:

1. The client calls method1 of the adapter implementation with the required input parameters.
2. The adapter converts the input request into a form expected by the adaptee.
3. The adapter then calls method2 of the adaptee, passing in the converted parameters, and receives a response.
4. The response, if required, is then converted to a form expected by the client before it is finally returned to the client itself.

The overall request flow is abstract to the client, who remains unaware of the adapter. It assumes that the response is received directly from the adaptee's implementation, while the request is first converted and then delegated to the correct interface implementation, all with the help of an adapter. This way, the adapter solves the problem of incompatibility among the interfaces and provides a clean approach to reuse the otherwise difficult-to-use interfaces.

The adapter can also be made to support more than one adaptee implementation with unique references pointing to various interface types if the features expected by the client are not served by a single interface. In such a case, the adapter implementation must have references to each type of interface that together provide all the features expected by the client.

Figure 7.2 exhibits the use of a single adapter to integrate multiple interfaces:

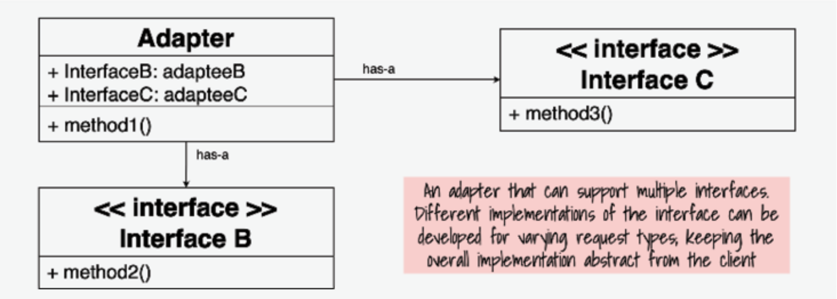

Adapter
+ InterfaceB: adapteeB
+ InterfaceC: adapteeC
+ method1()
has-a
<< interface >>
Interface C
+ method3()
has-a
<< interface >>
Interface B
+ method2()
An adapter that can support multiple interfaces. Different implementations of the interface can be developed for varying request types, keeping the overall implementation abstract from the client

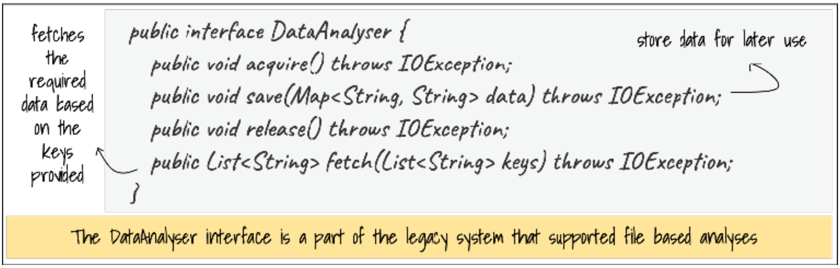
***Figure 7.2**: Adapter supporting multiple interfaces*

Building an adapter

The example that we will walk you through in this chapter makes use of a data analyzer that uses both modern and legacy systems to store and fetch data to perform analysis. We will learn how the adapter design pattern can be utilized to integrate a legacy implementation with a modern one to make it usable with modern-day clients who are incompatible with the legacy system. Let us first look at the legacy interface before the modern interface that supports the NoSQL-driven data storage mechanism.

Interfaces

The interface shown in *Figure 7.3* is part of a legacy, file-based data storage system that provides certain methods to support data insertion and read directly from the file system. This interface provides us with two additional methods apart from the usual save and fetch methods to store and fetch the record entries from the file system. Those two methods are acquire and release, and as their names suggest, they are used to acquire and release the file resource in a multithreaded environment.



```
public interface DataAnalyser {
    public void acquire() throws IOException;
    public void save(Map<String, String> data) throws IOException;
    public void release() throws IOException;
    public List<String> fetch(List<String> keys) throws IOException;
}
```

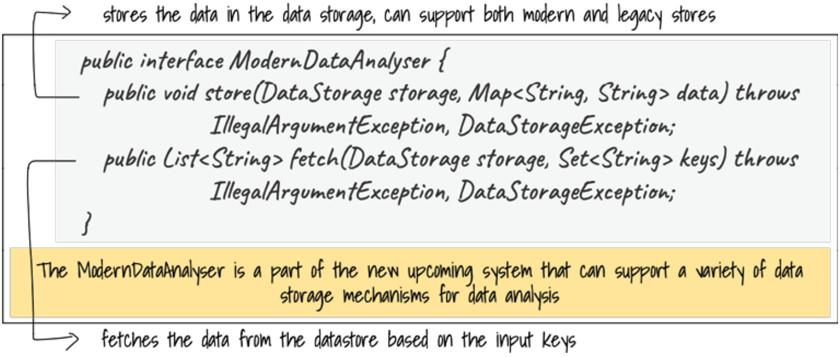
fetches the required data based on the keys provided

store data for later use

The DataAnalyser interface is a part of the legacy system that supported file based analyses

***Figure 7.3**: DataAnalyser interface*

The interface shown in *Figure 7.4* is a part of the new, NoSQL-based data storage system that provides certain methods to support data insertion and read from the distributed NoSQL database system, very similar to what the previous interface did. It is also the interface that is expected by the client. The client understands the method signatures and has logic written based on them.



stores the data in the data storage, can support both modern and legacy stores

```
public interface ModernDataAnalyser {
    public void store(DataStorage storage, Map<String, String> data) throws
                IllegalArgumentException, DataStorageException;
    public List<String> fetch(DataStorage storage, Set<String> keys) throws
                IllegalArgumentException, DataStorageException;
}
```

The ModernDataAnalyser is a part of the new upcoming system that can support a variety of data storage mechanisms for data analysis

fetches the data from the datastore based on the input keys

***Figure 7.4**: Interface of ModernDataAnalyser*

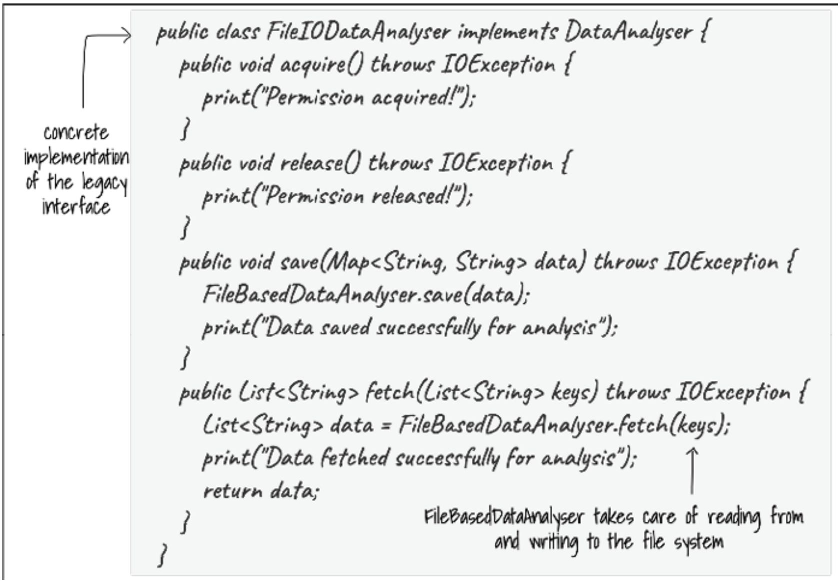
Both these interfaces perform similar tasks but provide a different set of methods. While the new interface is understood well by the client, the legacy is not.

In order to make use of the legacy interface and its implementations and to make it compatible with the client, we can write a wrapper class around the legacy interface that encapsulates its implementation and takes care of the input and output translations for us. This wrapper class must implement the interface that is understood by the client and must also ensure that it bridges the communication gap with the legacy interface.

The wrapper class that we just discussed is an adapter; it is an implementation of the type the client understands. It wraps the legacy interface implementation, giving it a new face, a face that can be integrated with the client.

Concrete implementations

Let us now explore some of the implementations for both the legacy and the modern interface. *Figure 7.5* exhibits an implementation of the legacy interface and provides access to the file-based data storage:



```
public class FileIODataAnalyser implements DataAnalyser {
   public void acquire() throws IOException {
      print("Permission acquired!");
   }
   public void release() throws IOException {
      print("Permission released!");
   }
   public void save(Map<String, String> data) throws IOException {
      FileBasedDataAnalyser.save(data);
      print("Data saved successfully for analysis");
   }
   public List<String> fetch(List<String> keys) throws IOException {
      List<String> data = FileBasedDataAnalyser.fetch(keys);
      print("Data fetched successfully for analysis");
      return data;
   }
}
```

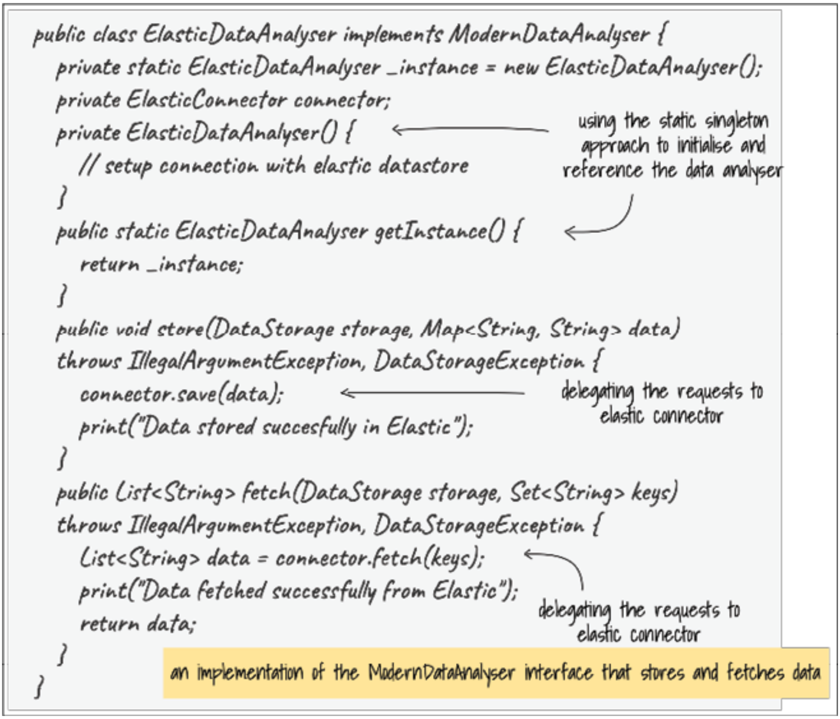
concrete implementation of the legacy interface

FileBasedDataAnalyser takes care of reading from and writing to the file system

***Figure 7.5**: FileIODataAnalyser*

The implementation is not highly productive as it does not show us how exactly we can use the file system to store and fetch data. To keep it short and simple, we are only focusing on the design aspects.

Figure 7.6 exhibits an implementation of the new interface and provides access to the NoSQL database for data storage and retrieval. This implementation is compatible with the client system and does not require being wrapped by an adapter; although in our example, we will wrap a reference to this implementation as well to showcase how a single adapter can work with multiple different implementations.



```
public class ElasticDataAnalyser implements ModernDataAnalyser {
   private static ElasticDataAnalyser _instance = new ElasticDataAnalyser();
   private ElasticConnector connector;
   private ElasticDataAnalyser() {
      // setup connection with elastic datastore
   }
   public static ElasticDataAnalyser getInstance() {
      return _instance;
   }
   public void store(DataStorage storage, Map<String, String> data)
   throws IllegalArgumentException, DataStorageException {
      connector.save(data);
      print("Data stored succesfully in Elastic");
   }
   public List<String> fetch(DataStorage storage, Set<String> keys)
   throws IllegalArgumentException, DataStorageException {
      List<String> data = connector.fetch(keys);
      print("Data fetched successfully from Elastic");
      return data;
   }
}
```

using the static singleton approach to initialise and reference the data analyser

delegating the requests to elastic connector

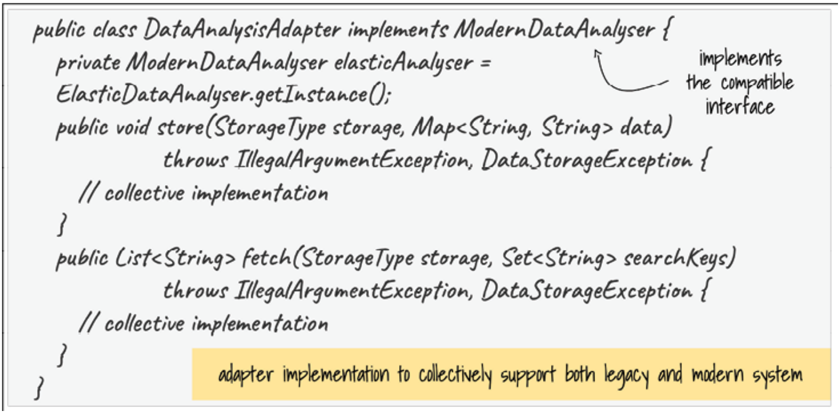
delegating the requests to elastic connector

an implementation of the ModernDataAnalyser interface that stores and fetches data

Figure 7.6: *ElasticDataAnalyser*

Adapter

The adapter implementation acts as a bridge between two incompatible interfaces. It implements one interface and acts as a wrapper to another. In *Figure 7.7*, the adapter will act as a single point of interaction for the client as it wraps both the legacy and modern implementations:



```
public class DataAnalysisAdapter implements ModernDataAnalyser {
   private ModernDataAnalyser elasticAnalyser =
   ElasticDataAnalyser.getInstance();
   public void store(StorageType storage, Map<String, String> data)
               throws IllegalArgumentException, DataStorageException {
      // collective implementation
   }
   public List<String> fetch(StorageType storage, Set<String> searchKeys)
               throws IllegalArgumentException, DataStorageException {
      // collective implementation
   }
}
```

implements the compatible interface

adapter implementation to collectively support both legacy and modern system

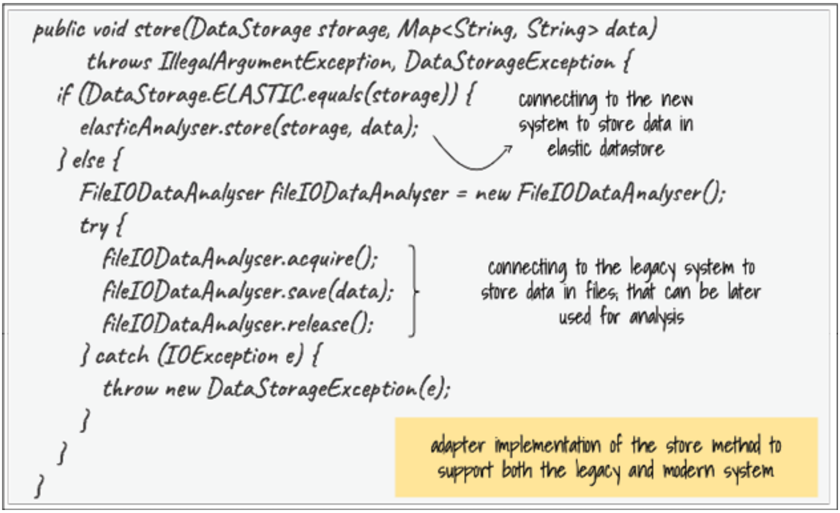
Figure 7.7: *Adapter implementation I*

It is not required to support the implementations that are compatible with each other. The adapters are supposed to make incompatible interfaces compatible so that they can be integrated with the rest of the system.

The reason we have encapsulated multiple interface implementations within our adapter is just to showcase that one adapter can be used with multiple interface implementations, and because the modern interface has the **StorageType** parameter that provides the type to be used for the incoming request. If there is no way to distinguish that, it will not be possible for the adapter to support multiple interface implementations.

Now, let us look at the two method definitions that perform the task of translating the data before delegating the request to actual implementations. The store method, for the elastic analyzer, does not perform any special action and simply delegates the call to the actual implementation. However, for the file-based analyzer, the **store** method makes some additional calls to the legacy implementation apart from calling the **save** method to ensure the correctness of its behavior. This allows for adapting to and bridging the gap between the client and legacy interfaces.

Figure 7.8 demonstrates the implementation of the store method that supports both the legacy and modern systems:



```
public void store(DataStorage storage, Map<String, String> data)
      throws IllegalArgumentException, DataStorageException {
  if (DataStorage.ELASTIC.equals(storage)) {
    elasticAnalyser.store(storage, data);
  } else {
    FileIODataAnalyser fileIODataAnalyser = new FileIODataAnalyser();
    try {
      fileIODataAnalyser.acquire();
      fileIODataAnalyser.save(data);
      fileIODataAnalyser.release();
    } catch (IOException e) {
      throw new DataStorageException(e);
    }
  }
}
```

connecting to the new system to store data in elastic datastore

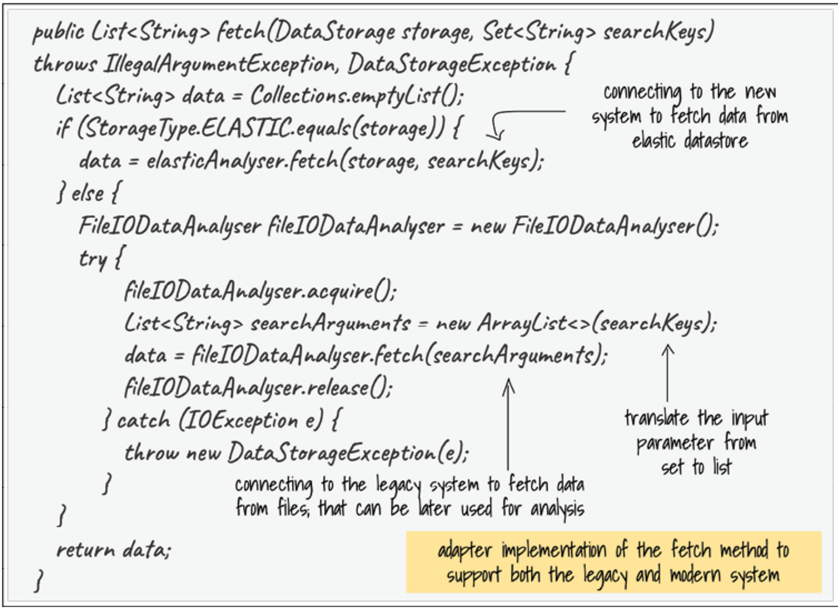
connecting to the legacy system to store data in files, that can be later used for analysis

adapter implementation of the store method to support both the legacy and modern system

***Figure 7.8**: Adapter implementation II*

If you are thinking about the translation of the input and the output, then it is not required because the method signatures for both interfaces expect the same input parameters. The fetch method, on the other hand, requires input translation from one form to another to support the method signature of the legacy interface.

Figure 7.9 exhibits the implementation of the fetch method that supports both the legacy and modern systems. If you had noticed, the exceptions thrown by the legacy interface methods are also different from the ones expected by the client.



```
public List<String> fetch(DataStorage storage, Set<String> searchKeys)
throws IllegalArgumentException, DataStorageException {
   List<String> data = Collections.emptyList();
   if (StorageType.ELASTIC.equals(storage)) {
      data = elasticAnalyser.fetch(storage, searchKeys);
   } else {
      FileIODataAnalyser fileIODataAnalyser = new FileIODataAnalyser();
      try {
            fileIODataAnalyser.acquire();
            List<String> searchArguments = new ArrayList<>(searchKeys);
            data = fileIODataAnalyser.fetch(searchArguments);
            fileIODataAnalyser.release();
        } catch (IOException e) {
            throw new DataStorageException(e);
        }
   }
   return data;
}
```

connecting to the new system to fetch data from elastic datastore

translate the input parameter from set to list

connecting to the legacy system to fetch data from files; that can be later used for analysis

adapter implementation of the fetch method to support both the legacy and modern system

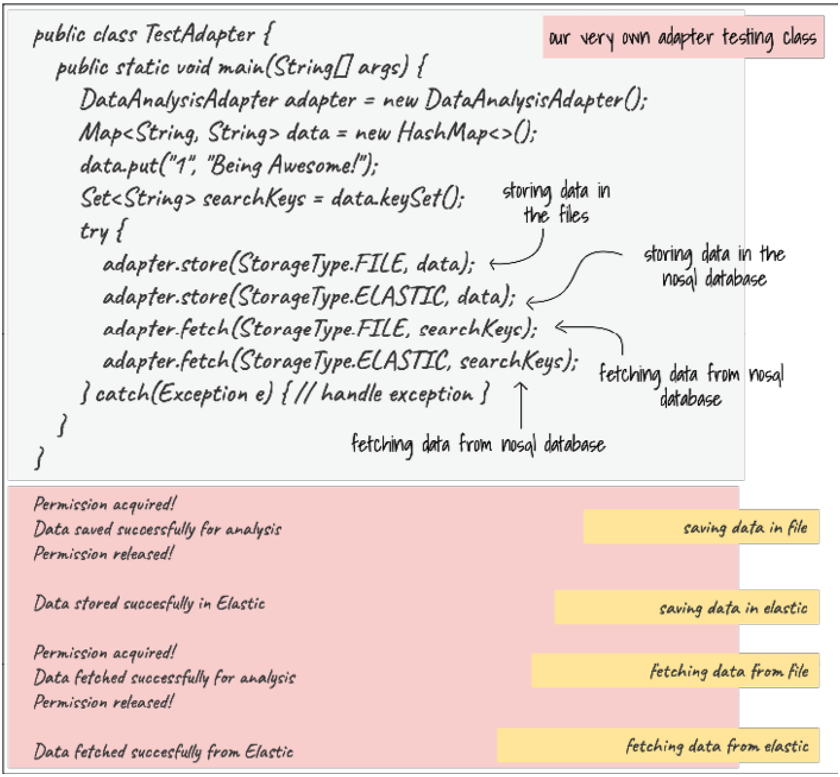
Figure 7.9: *Adapter implementation III*

In short, the adapter is responsible for ensuring that the client is unaware of its existence.

Testing the adapter

We have written a test class to verify the working of the adapter design pattern. Let us explore the same before we move to the next section.

In *Figure 7.10*, we call the adapter implementation, passing in the correct storage type and letting the adapter take care of the translation and compatibility. Based on the storage type provided, the adapter calls the relevant implementation to perform the requested operation.



```
public class TestAdapter {
  public static void main(String[] args) {
    DataAnalysisAdapter adapter = new DataAnalysisAdapter();
    Map<String, String> data = new HashMap<>();
    data.put("1", "Being Awesome!");
    Set<String> searchKeys = data.keySet();
    try {
      adapter.store(StorageType.FILE, data);
      adapter.store(StorageType.ELASTIC, data);
      adapter.fetch(StorageType.FILE, searchKeys);
      adapter.fetch(StorageType.ELASTIC, searchKeys);
    } catch(Exception e) { // handle exception }
  }
}
```

```
Permission acquired!
Data saved successfully for analysis
Permission released!

Data stored succesfully in Elastic

Permission acquired!
Data fetched successfully for analysis
Permission released!

Data fetched succesfully from Elastic
```

our very own adapter testing class
storing data in the files
storing data in the nosql database
fetching data from nosql database
fetching data from nosql database
saving data in file
saving data in elastic
fetching data from file
fetching data from elastic

***Figure 7.10**: Test adapter implementation*

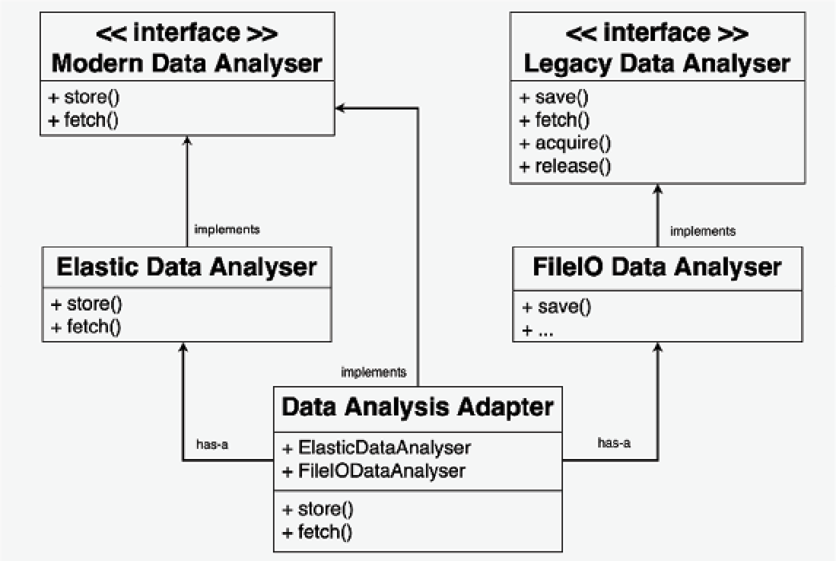
The preceding example exhibits how an adapter implementation can be used to work with multiple interface implementations.

Summary

The adapter design pattern, as the name implies, acts as an adapter to make two or more interfaces compatible with each other. It allows us to make interfaces compatible with each other. The adapter class implements one of the interfaces and wraps the other to form a communication channel between the two interfaces so that the two can work together, which otherwise is not possible. All this is done just by introducing an adapter class and without modifying any of the existing code, be it the incompatible interfaces or the client.

In *Figure 7.11*, the **DataAnalysisAdapter** class implements the

ModernDataAnalyser interface and wraps **FileIODataAnalyser**, so that the client can directly connect to and make use of the services provided by the legacy interface.



<< interface >>
Modern Data Analyser
+ store()
+ fetch()
<< interface >>
Legacy Data Analyser
+ save()
+ fetch()
+ acquire()
+ release()
implements
implements
Elastic Data Analyser
+ store()
+ fetch()
FileIO Data Analyser
+ save()
+ ...
implements
Data Analysis Adapter
+ ElasticDataAnalyser
+ FileIODataAnalyser
+ store()
+ fetch()
has-a
has-a

***Figure 7.11**: Adapter design pattern defined*

The adapter class in our example has a reference to both incompatible types and allows for converting the request from one form to another, respecting both the caller and the callee.

Advantages and drawbacks

It is time to make note of some of the benefits and drawbacks of the adapter design pattern. Let us start with the benefits first followed by the drawbacks:

- It provides a way to integrate incompatible interfaces.
- It allows the reuse of existing functionality.
- It requires little or no change in the existing interfaces and their concrete implementations.
- It allows the separation of the integration code from the primary business logic.

The drawbacks of the adapter design pattern are:

- Adapter cannot be used with subclasses.
- The overall complexity of the code is increased because of the introduction of adapter classes.

Usage

The following list outlines some of the scenarios that are relevant for constructing adapters:

- When the requirement is to reuse existing code that has an incompatible interface and requires integration.
- When two or more incompatible interfaces must communicate to perform a common task.

Conclusion

In this chapter, we learned that an adapter design pattern is a widely used structural design pattern that can be utilized to make incompatible interfaces compatible without making many changes. The design pattern helps to reuse the existing code, which otherwise would have to be rewritten every time there is a client that needs its services but cannot because of the difference in their interfaces. The adapter class performs two basic tasks: It translates the input and output as required by the two adapting interfaces, and it implements the interface that is compatible with the client and wraps the implementation of the incompatible interface.

In the next chapter, we will discuss another structural design pattern known as the **decorator design pattern** and will learn another way to wrap existing code to provide new features.

Questions

In this section, we will sharpen our brains by indulging in some fun brain activities. Let us get started.

Find the odd one out

1. Wrapper, adapter, adaptee, client
2. Code reuse, signature modification, introduce an adapter,

minimal changes

3. Incompatible interfaces, adaptee, no modification, adapter
4. Adapter design pattern
    a. Makes incompatible interfaces compatible
    b. Require modifications in the existing code
    c. Introduces a wrapper class
    d. Promotes code reuse
5. An adapter design pattern can be used when
    a. There is a need to integrate incompatible interfaces
    b. Code rewrite is not desired
    c. Classes are wrapped for performance improvement
    d. Existing interfaces are to be integrated
6. The adapter design pattern has these important actors
    a. An adapter
    b. An adaptee
    c. Incompatible interfaces
    d. Compatible interfaces

Select two correct answers

1. Adapter pool design pattern
    a. Integrates incompatible interfaces
    b. Introduces a lot of changes in the existing code
    c. Works well with legacy architecture
    d. Can only be used for compatible interfaces
2. Main actors in the adapter design pattern
    a. Adapter
    b. Adaptee
    c. Singleton
    d. Worker
3. What goes together
    a. Adapter, adaptee, incompatible interfaces

b. Code reuse, compatible interfaces, adapter
c. Adapter, code reuse, adaptee
d. Incompatible interfaces, adapters, compatible interfaces

Answers

This section provides the answers to the questions given.

Find the odd one out

1. Wrapper
2. Signature modification
3. No modification
4. Require modifications in existing code
5. Classes are wrapped for performance improvement
6. Compatible interfaces

Select two correct answers

1. Integrates incompatible interfaces, works well with legacy architecture
2. Adapter, adaptee
3. Adapter, adaptee, incompatible interfaces, and adapter, code reuse, adaptee

CHAPTER 8

Decorating Objects

Introduction

When we think of extending the functionality of a class, direct modification and subclassing are the first two approaches that are essential. By altering a class structure or by extending a class, we can override the existing behavior of a type or add additional behavior to it. It might appear straightforward, but there are some caveats to it; with sub-classing, like you cannot extend more than one class, and every time there is a need for additional behavior, you end up doing the same all over again, and not all classes are open for direct modification as well.

There is another approach that can be used to add behavior to class objects, without any modification to the existing code. By wrapping objects and binding them with a has-a relationship, objects can be reused to support additional behavior without making any changes to them. In this chapter, we will learn about the decorator design pattern and how it utilizes a has-a relationship to create decorator objects that enhance the functionality of the original object without altering its structure.

Structure

This chapter covers the following topics:

- Wrappers
- The decorator design pattern
- Summary
- Advantages and drawbacks
- Usage

Objectives

By the end of this chapter, the readers will develop a thorough understanding of the use of wrappers to add behavior to objects dynamically. They will learn about the decorator design pattern and how to add behavior to objects without making any changes to their structure. They will also learn about the decorator chain and how it helps with ensuring the principle of the separation of concerns.

Readers will also discuss the implementation details of the decorator design pattern, the benefits it provides, and a few drawbacks, along with some of the places where this design pattern can be used.

Wrappers

All this time, we have been looking at this problem from the top when we should be looking at it from the bottom. Instead of altering or extending a type, we can simply wrap it with another class that follows the same behavioral contract.

Figure 8.1 exhibits the class hierarchy in object-oriented languages:

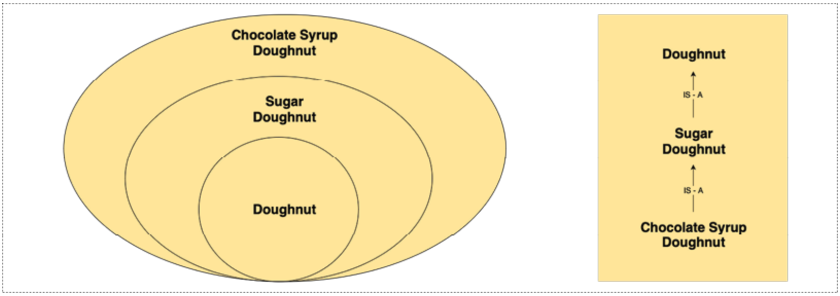


***Figure 8.1**: Class hierarchy*

By using wrapper classes, we can provide additional behavior to a type while ensuring that the existing behavior is preserved. *Figure 8.2* exhibits the use of composition to support additional functionality:

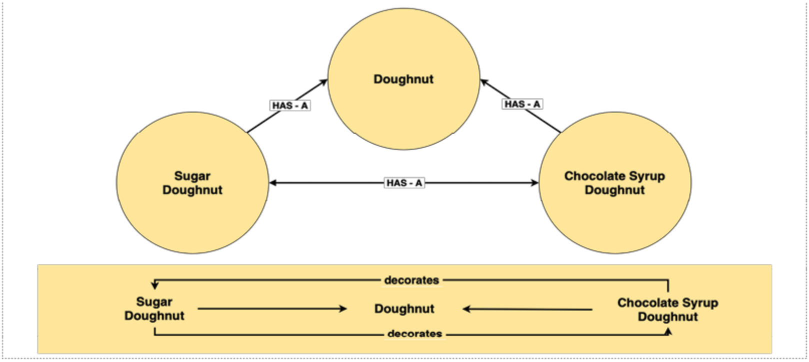


***Figure 8.2**: Wrapper objects*

When a class extends another, all the objects of the child class share the same behavior, but wrapping allows us to add unique behaviors to an object without affecting any other objects of that type.

The decorator design pattern

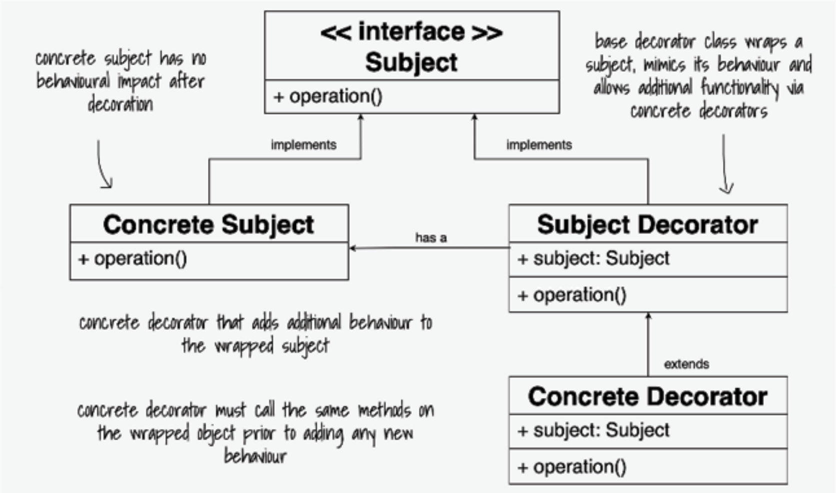
The decorator design pattern is a structural design pattern that allows adding some extra behavior to an object without altering the behavior of other objects of that type. It makes use of decorator classes that implement the same interface as the subject class. The decorator object wraps the subject, that is, the original object, to provide additional behavior or to enhance its current behavior.

For adding behavior to a class, the usual consideration is either to alter its structure or to subclass or extend it. The decorators are an alternative to that approach that do not require any modification in the subject class. Due to the dynamic addition of behavior, the decorator design pattern is very useful when we do not have access to the subject classes.

The decorator design pattern can also be used to enhance the behavior of objects that, by definition, are not open for extension, that is, immutable objects. Such objects can be wrapped by a decorator to provide additional behavior.

By design, a decorator can wrap another decorator and form a chain of additional behaviors. This allows segregation of logic into multiple decorators and promotes the single responsibility principle and the principle of separation of concerns.

In *Figure 8.3*, the **Decorator** implements the **Subject** interface like a concrete implementation:



<< interface >>
Subject
+ operation()
concrete subject has no behavioural impact after decoration
base decorator class wraps a subject, mimics its behaviour and allows additional functionality via concrete decorators
implements
implements
Concrete Subject
+ operation()
has a
Subject Decorator
+ subject: Subject
+ operation()
concrete decorator that adds additional behaviour to the wrapped subject
extends
concrete decorator must call the same methods on the wrapped object prior to adding any new behaviour
Concrete Decorator
+ subject: Subject
+ operation()

***Figure 8.3**: Decorator design pattern*

While the concrete implementation is itself of type **Subject** and so is the **Decorator**, the decorator class also has a reference to the interface type. This reference plays a vital role in the decorator design pattern; it could be assigned an object of that type that, in turn, is decorated with additional behavior.

Refer to the following list, which explains its working:

1. An object of type **Subject** is instantiated.
2. A decorator is instantiated that wraps the subject.
3. When a method is called on the decorator instance, it calls the same method on the wrapped object and appends any additional behavior written in the decorator.

The decorator appends additional behavior to the method call without altering the existing behavior of the wrapped object. By creating more such decorators, multiple behaviors can be written to add to the default behavior of the concrete subject implementation.

The decorators can wrap any object of the interface type, which includes the decorators themselves, opening the prospect of appending multiple behaviors on top of each other in a stack-like manner. There is no specific order in which the decorators can be used, and thus different objects can be made to behave differently following the sequence in which the decorators are arranged to decorate the subject.

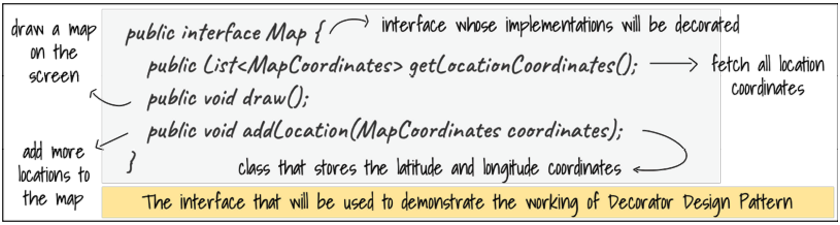
Let us look at an example to add clarity to our understanding of the decorator design pattern.

Building a decorator

The example that we will discuss uses a map interface, a concrete implementation of it, and a few decorators. We will learn how the decorator design pattern can be utilized to append additional behavior to existing object instances. We will also learn about the decorator chain that can be formed by wrapping decorators within decorators and the way it can be used to append behaviors in many different orders. Let us first look at the interface that sets up the common contract that both the concrete implementations and the decorators must follow for the design to work.

Interface

The interface is the central point of the decorator design pattern. In *Figure 8.4*, the **Map** interface provides the behavior that must be implemented by the concrete implementations as well as the decorators.



```
public interface Map {
    public List<MapCoordinates> getLocationCoordinates();
    public void draw();
    public void addLocation(MapCoordinates coordinates);
}
```

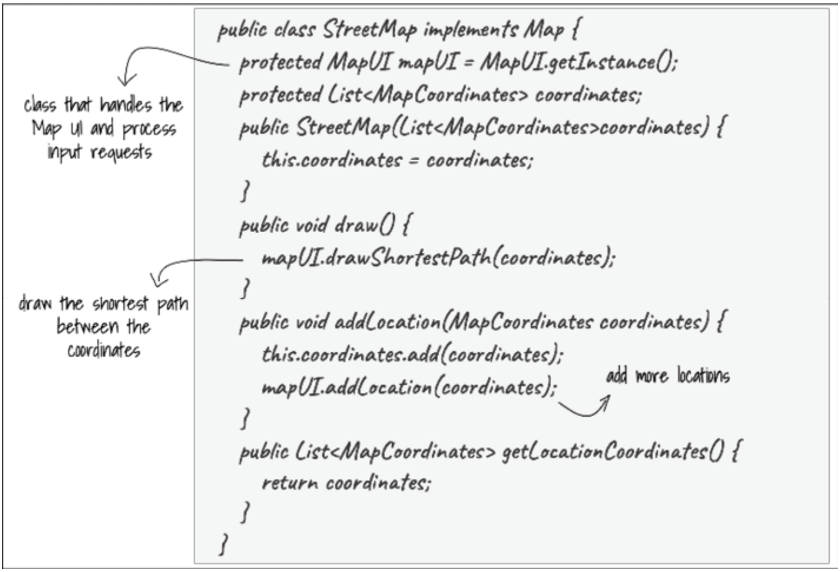
draw a map on the screen
interface whose implementations will be decorated
fetch all location coordinates
add more locations to the map
class that stores the latitude and longitude coordinates
The interface that will be used to demonstrate the working of Decorator Design Pattern

***Figure 8.4**: The Map interface*

This interface provides us with three methods that can be used to draw a map, add more locations on the map, and fetch all the location coordinates.

The subject

Figure 8.5 exhibits a concrete implementation of the interface as shown in *Figure 8.4*. This concrete implementation is the **subject** of the decorator design pattern. The decorators that we will construct in the following section of this chapter will wrap an instance of our subject to provide additional behavior on top of what is already provided by the subject.



```
public class StreetMap implements Map {
  protected MapUI mapUI = MapUI.getInstance();
  protected List<MapCoordinates> coordinates;
  public StreetMap(List<MapCoordinates>coordinates) {
    this.coordinates = coordinates;
  }
  public void draw() {
    mapUI.drawShortestPath(coordinates);
  }
  public void addLocation(MapCoordinates coordinates) {
    this.coordinates.add(coordinates);
    mapUI.addLocation(coordinates);
  }
  public List<MapCoordinates> getLocationCoordinates() {
    return coordinates;
  }
}
```

class that handles the Map UI and process input requests
draw the shortest path between the coordinates
add more locations

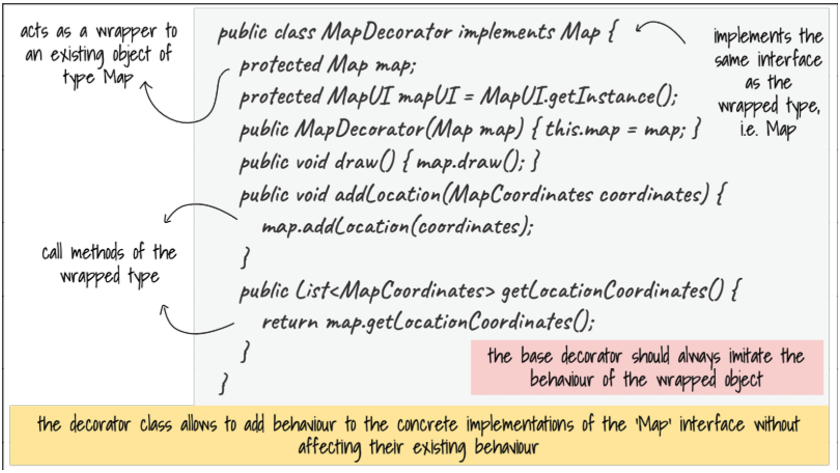
Figure 8.5: *StreetMap | Concrete implementation*

This is just one of the implementations of the interface, and there could be many more based on the rendering required for different scenarios, for example, street view, satellite view, terrain view, and so on.

Base decorator

The base decorator and the subject implement the same interface, but the decorator is more than just a concrete implementation. It wraps an object of the same type as that of the interface it implements. It does not have any behavior of its own and delegates all the incoming requests to the object it wraps. This is the most crucial aspect of the base decorator.

Figure 8.6 exhibits a base decorator. The decorator class is structured in a manner to decorate whichever object is passed to it, be it a concrete implementation of that type or another decorator that implements the same interface.



```
public class MapDecorator implements Map {
  protected Map map;
  protected MapUI mapUI = MapUI.getInstance();
  public MapDecorator(Map map) { this.map = map; }
  public void draw() { map.draw(); }
  public void addLocation(MapCoordinates coordinates) {
    map.addLocation(coordinates);
  }
  public List<MapCoordinates> getLocationCoordinates() {
    return map.getLocationCoordinates();
  }
}
```

acts as a wrapper to an existing object of type Map

implements the same interface as the wrapped type, i.e. Map

call methods of the wrapped type

the base decorator should always imitate the behaviour of the wrapped object

the decorator class allows to add behaviour to the concrete implementations of the 'Map' interface without affecting their existing behaviour

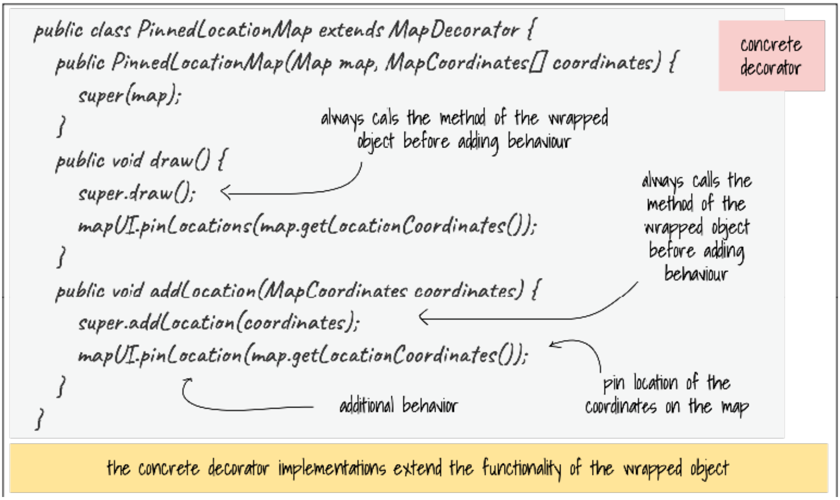
***Figure 8.6**: Map decorator*

The method definitions in the base decorator mimic the behavior of the **subject** passed to it. You must be thinking, *why even have a base decorator when all it does is just call methods of some wrapped object, and when the concrete implementations could have done the same as well?* The decorators are not supposed to override all the methods, but only those that must support additional behavior. A base decorator helps here by providing default implementations to every method of the interface, effectively reducing the need for the concrete ones to implement them. Apart from the methods provided by the interface, the decorators can also have their own methods that can provide features missing in the subject class, the one we wrap with decorators.

Let us now look into some of the decorators and understand the overall working of the decorator design pattern.

Concrete decorators

Figure 8.7 and *Figure 8.8* exhibit two concrete decorators that extend the base decorator and override the relevant methods. The overridden methods make calls to the parent decorator to make sure that the default implementation is always executed. *Figure 8.7* exhibits a decorator that pins the locations on the map both when the map is drawn and when a new location is added to it:



```
public class PinnedLocationMap extends MapDecorator {
   public PinnedLocationMap(Map map, MapCoordinates[] coordinates) {
      super(map);
   }
   public void draw() {
      super.draw();
      mapUI.pinLocations(map.getLocationCoordinates());
   }
   public void addLocation(MapCoordinates coordinates) {
      super.addLocation(coordinates);
      mapUI.pinLocation(map.getLocationCoordinates());
   }
}
```

concrete decorator

always cals the method of the wrapped object before adding behaviour

always calls the method of the wrapped object before adding behaviour

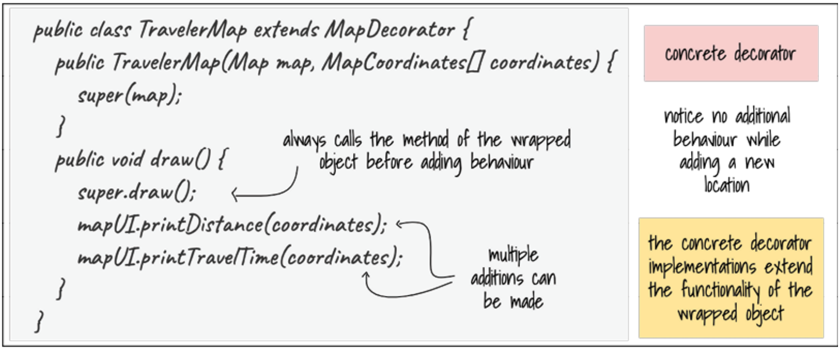
pin location of the coordinates on the map

additional behavior

the concrete decorator implementations extend the functionality of the wrapped object

Figure 8.7: *PinnedLocationMap decorator*

Figure 8.8 exhibits a decorator that prints the distance between locations as well as the time required to travel that distance when the map is drawn. Only one of the methods is overridden, and the base decorator provides the definition for the other.



```
public class TravelerMap extends MapDecorator {
   public TravelerMap(Map map, MapCoordinates[] coordinates) {
      super(map);
   }
   public void draw() {
      super.draw();
      mapUI.printDistance(coordinates);
      mapUI.printTravelTime(coordinates);
   }
}
```

concrete decorator

always calls the method of the wrapped object before adding behaviour

notice no additional behaviour while adding a new location

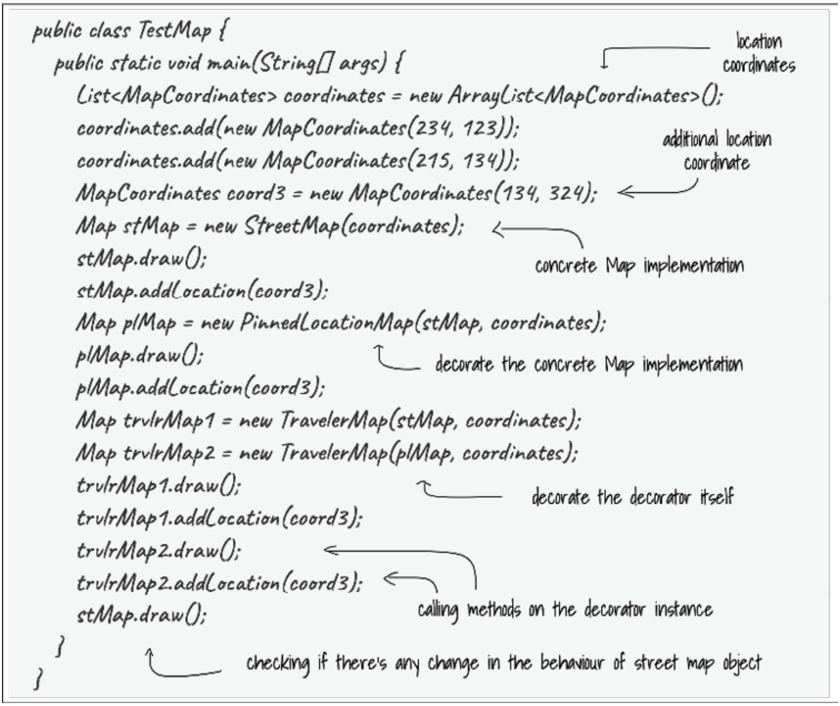
multiple additions can be made

the concrete decorator implementations extend the functionality of the wrapped object

Figure 8.8: *TravelerMap decorator*

Testing the decorators

We have written a test class to verify the working of the decorator design pattern. Let us execute the same as shown in *Figure 8.9*:



```
public class TestMap {
  public static void main(String[] args) {
    List<MapCoordinates> coordinates = new ArrayList<MapCoordinates>();
    coordinates.add(new MapCoordinates(234, 123));
    coordinates.add(new MapCoordinates(215, 134));
    MapCoordinates coord3 = new MapCoordinates(134, 324);
    Map stMap = new StreetMap(coordinates);
    stMap.draw();
    stMap.addLocation(coord3);
    Map plMap = new PinnedLocationMap(stMap, coordinates);
    plMap.draw();
    plMap.addLocation(coord3);
    Map trvlrMap1 = new TravelerMap(stMap, coordinates);
    Map trvlrMap2 = new TravelerMap(plMap, coordinates);
    trvlrMap1.draw();
    trvlrMap1.addLocation(coord3);
    trvlrMap2.draw();
    trvlrMap2.addLocation(coord3);
    stMap.draw();
  }
}
```

location coordinates

additional location coordinate

concrete Map implementation

decorate the concrete Map implementation

decorate the decorator itself

calling methods on the decorator instance

checking if there's any change in the behaviour of street map object

Figure 8.9: *Test decorators*

In *Figure 8.9*, we start by setting up the map coordinates and instantiating an instance of our subject, **StreetMap**. Then we instantiate two decorator objects, one each of type **PinnedLocationMap** and **TravelerMap**. We then execute the methods on each of these instances to verify the behavioral change when the same methods are called on the subject and then on the decorators. We have also decorated a decorator to establish a chain of decorators and to confirm that it works. In *Figure 8.10*, you can see the results of calling the same methods on the subject as well as the decorator objects:

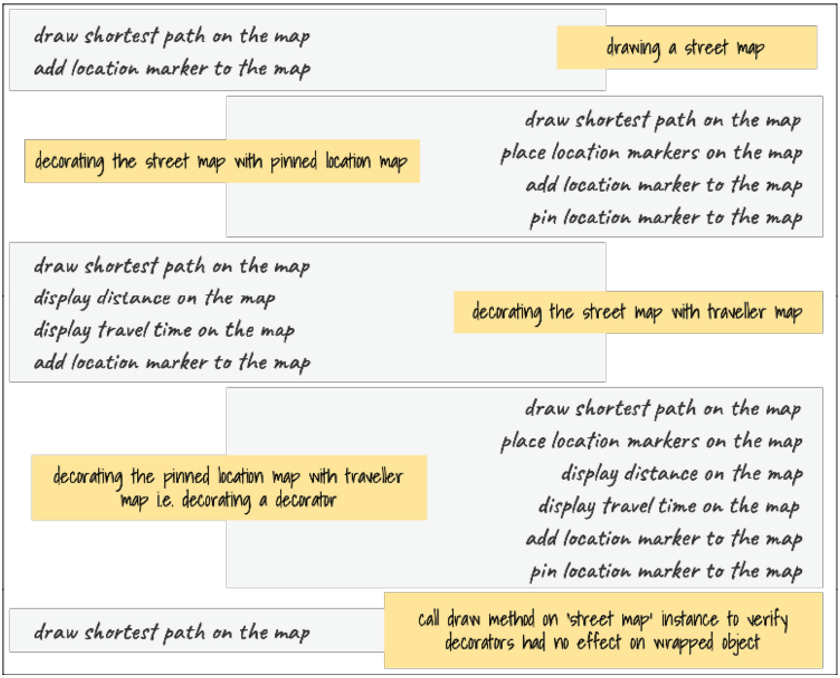
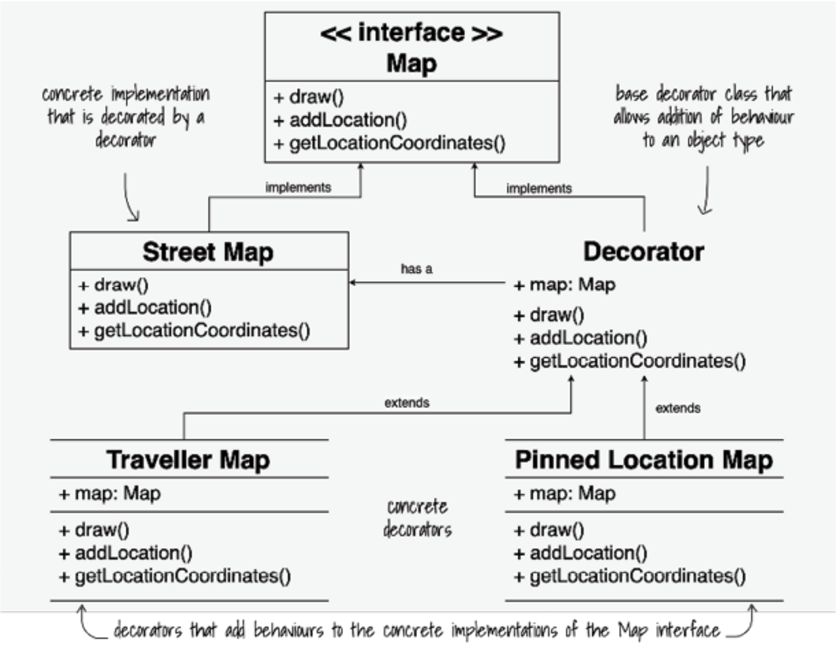

draw shortest path on the map
add location marker to the map
drawing a street map
decorating the street map with pinned location map
draw shortest path on the map
place location markers on the map
add location marker to the map
pin location marker to the map
draw shortest path on the map
display distance on the map
display travel time on the map
add location marker to the map
decorating the street map with traveller map
decorating the pinned location map with traveller map i.e. decorating a decorator
draw shortest path on the map
place location markers on the map
display distance on the map
display travel time on the map
add location marker to the map
pin location marker to the map
draw shortest path on the map
call draw method on 'street map' instance to verify decorators had no effect on wrapped object

***Figure 8.10**: Test decorator result*

First, we executed methods on the subject instance, then by wrapping that instance in two different decorators, and then we decorated the decorator itself. In the end, the **draw** method is executed again on the same subject instance to verify that there is no change in its behavior and that the decorators had no effect on it.

Summary

Figure 8.11 exhibits the architecture of the decorator design pattern. The decorator design pattern, as the name implies, decorates another object to provide additional behavior on top of what was provided by that object. A decorator has no impact on the structure of the subject. The decorators implement the same interface as the subject and wrap an object of the interface type to provide additional behavior.



<< interface >>
Map
+ draw()
+ addLocation()
+ getLocationCoordinates()
concrete implementation that is decorated by a decorator
base decorator class that allows addition of behaviour to an object type
implements
implements
Street Map
+ draw()
+ addLocation()
+ getLocationCoordinates()
has a
Decorator
+ map: Map
+ draw()
+ addLocation()
+ getLocationCoordinates()
extends
extends
Traveller Map
+ map: Map
+ draw()
+ addLocation()
+ getLocationCoordinates()
concrete decorators
Pinned Location Map
+ map: Map
+ draw()
+ addLocation()
+ getLocationCoordinates()
decorators that add behaviours to the concrete implementations of the Map interface

***Figure 8.11**: Decorator design pattern—defined*

The decorator design pattern is useful when the classes are not open for extension because they are final in nature or when multiple distinct behaviors are to be added.

Advantages and drawbacks

Let us discuss the benefits and drawbacks of the decorator design pattern. The benefits are as follows:

- It provides a way to add behavior to an object without altering the structure or existing behavior of that object.
- It can be used to stack together multiple behaviors on top of each other by chaining decorators together.
- It is better than using static inheritance.
- It provides a flexible way to add and revoke behavior dynamically.

The drawback of the decorator design pattern is that using too many decorators can be hard to maintain and must be carefully ordered when chained together.

Usage

The following list outlines some of the scenarios that are relevant to constructing decorators:

- Providing unique behaviors on top of a common, generic behavior, for example, converting the output generated by the subject into different formats.
- For chaining behaviors together while adding some and skipping others based on conditions.

Conclusion

Decorator design pattern is a widely used structural design pattern that can be utilized to decorate existing objects without altering their structure. The design pattern helps to add additional behaviors, and it can be utilized to form decorator chains to append distinct behaviors on top of each other while ensuring no change to the subject.

In the next chapter, we will discuss another structural design pattern known as the proxy design pattern and will learn how to create proxy objects that guard the real object.

Questions

In this section, we will sharpen our brains by indulging in some fun brain activities. Let us get started.

Find the odd one out

1. Decorator, chain, additional behavior, marker interface
2. Code reuse, base interface, structure alteration, behavior design
3. Subject, decorator, not open for extension, no available interface
4. Decorator design pattern:
    a. Decorates existing objects
    b. Requires modifications in the existing code

    - c. Introduces a decorator class
    - d. Promotes code reuse
5. Decorate design pattern can be used when:
    - a. Additional behavior is desired
    - b. There is no base interface
    - c. Subject class is not open for change
    - d. Class is final in nature
6. The adapter design pattern has these important actors:
    - a. A decorator
    - b. A subject
    - c. A base interface
    - d. A polymorphic reference

Select two correct answers

1. Decorator design pattern
    - a. Decorates existing objects
    - b. Requires modifications in the existing code
    - c. Requires a base interface to implement
    - d. Can only be used with non-final classes
2. Decorator design pattern is most useful when
    - a. Class is not open for extension
    - b. Subject's source code is not available
    - c. Sub-classing is the only solution
    - d. Performance is the key
3. What goes together
    - a. Subject, decorator, final
    - b. Decorator chain, multiple behaviors, decorator
    - c. Decorator, sub-classing, inheritance
    - d. Wrapper, subject, no interface

Answers

This section provides the answers to the given questions.

Find the odd one out

1. Marker interface
2. Structure alteration
3. Not open for extension
4. Requires modifications in existing code
5. Subject class is not open for change
6. A polymorphic reference

Select two correct answers

1. Decorates existing objects, requires a base interface to implement
2. Class is not open for extension, subject’s source code is not available
3. Subject, decorator, final and decorator chain, multiple behaviors, decorator

CHAPTER 9

The Guardian

Introduction

The process of software development is a continuous one. A software provides many different functionalities that are often developed in iterations and are independent of each other, but not always. At times, we also need to add to an existing functionality. The approach to achieving the same seems straightforward by making the necessary changes directly to the existing class. This is only possible when the class is available and open for change. Also, we must consider the single responsibility principle and must not overburden the class with multiple functionalities.

If a class is not available for changes, if it is a part of some library we do not have direct access to, one option is to construct another class that provides the additional functionality and makes changes in the sequence of calls that lead the request to the original object. The approach could be a bit difficult to manage when the structure of the new class is different from the original one, as it could lead to integration issues with the classes that interact with the original class. *What if there is a class that resembles the same structure as that of the original one and that could be easily integrated with the rest of the* *code?*

In this chapter, we will learn about the proxy design pattern and how it can be utilized to create surrogate objects that guard the original object and may perform some actions before providing access to it.

Structure

This chapter covers the following topics:

- Proxies
- Summary
- Usage
- Build a proxy
- Adantages and drawbacks

Objectives

By the end of this chapter, the readers will be able to understand object proxies and how they can be utilized to control access to the objects they substitute. They will learn about the proxy design pattern and the different types of proxy objects that can be constructed to act as placeholders for the original objects.

The chapter will also walk them through the implementation details of the proxy design pattern, its benefits, and a few drawbacks, along with some of the places where this design pattern can be used.

Proxies

A proxy acts much like a placeholder, a substitute in place of some other object. It resembles the original object and can be easily integrated with the existing codebase. It implements the same interface as the original object and wraps it to serve as its proxy. It is often termed as a wrapper, a surrogate, or a substitute. *Figure 9.1* exhibits a proxy object and how it can be inserted in the call stack:

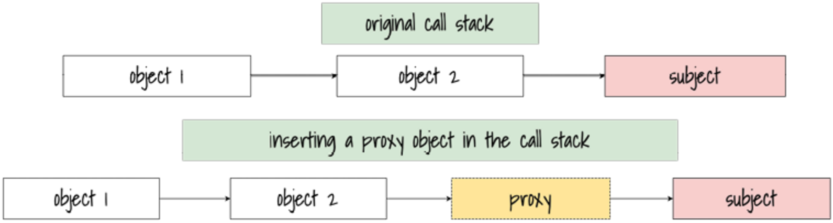


***Figure 9.1**: Inserting a proxy in the call stack*

Inserting a proxy into the call stack enables it to filter all the requests that reach the original object and perform additional operations. The original object instance can be easily replaced with a proxy instance without affecting the existing integration. These operations could be related to caching, logging, access check, marshaling, or something else, depending upon the requirement.

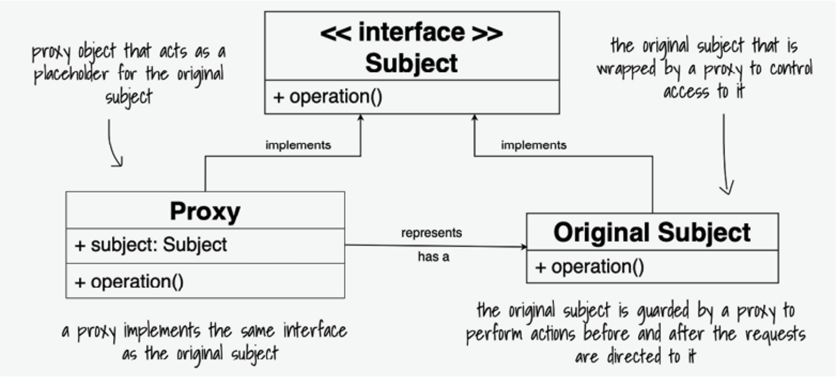
Proxy design pattern

The proxy design pattern is a structural design pattern that allows an object to serve as a placeholder for some other object. In general terms, it acts as a proxy to some interface that the client wants to access. The proxy object encapsulates the complexity of the object it wraps. It could also be described as a filter that intercepts all the incoming requests for the subject, performs some action before directing the request to the original object, and then serves the request is served by it.

Multiple proxies can be chained together to operate in a sequence, with each proxy performing a specific task. The order of execution depends on how these proxy objects wrap each other. The subject is wrapped by the last proxy in the sequence.

A prominent real-life example of a proxy is a manager and his assistant. Every request to meet the manager must go through his assistant. The assistant takes care of the manager’s calendar and who he would be interested in meeting.

In *Figure 9.2*, the **Proxy** implements the **Subject** interface like an original subject. The **proxy** class has a reference of type **Subject** that can be passed either the subject or another proxy object to form a chain.



proxy object that acts as a placeholder for the original subject
<< interface >>
Subject
+ operation()
the original subject that is wrapped by a proxy to control access to it
implements
implements
Proxy
+ subject: Subject
+ operation()
represents
has a
Original Subject
+ operation()
a proxy implements the same interface as the original subject
the original subject is guarded by a proxy to perform actions before and after the requests are directed to it

Figure 9.2: *Proxy design pattern*

Here is how it works:

1. Instantiate an object of the original type, a **Subject**.
2. Construct a proxy object and pass it to the **Subject**.
3. Direct all requests to the proxy object instead of the **Subject**.
4. When the proxy object receives a request, it performs some additional logic, along with delegating the request to the original object, and finally sends a response to the caller.

The proxy object does not impact the behavior of the original object, though it could alter the chain of calls in some cases. Multiple proxies can be chained together to perform various tasks working together.

The proxy design pattern can be utilized in many ways. It can be used to control access to an object, perform lazy initialization, cache data, audit the incoming requests, and much more.

Types of proxies

The proxy design pattern comes in several forms, each serving a distinct purpose. It is usually described as having specific types, like the remote proxy, which acts as a local representative of an object located in another address space or server; the virtual proxy, which delays the creation or loading of expensive resources until they are truly needed; and the protection (control) proxy, which enforces access rules or permissions before delegating a request.

Remote proxy

It is a proxy that represents a remote object. *Figure 9.3* exhibits an example of a remote proxy that encapsulates connectivity to the remote object, the serialization and deserialization required, the necessary access checks, and any other task that is required to connect to that remote object.

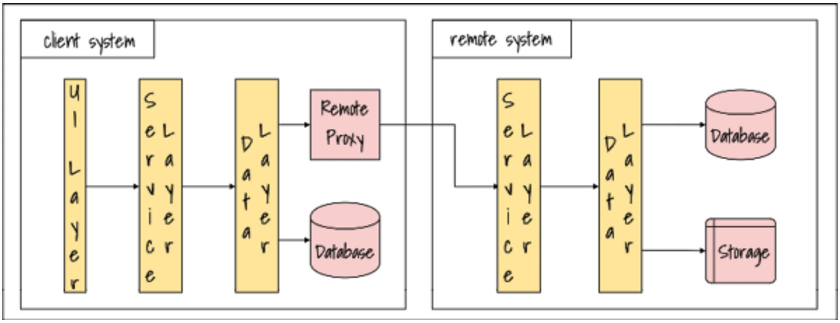

client system
UI Layer
Service Layer
Data Layer
Remote Proxy
Database
remote system
Service Layer
Data Layer
Database
Storage

Figure 9.3: *Remote proxy*

The remote proxy object is a placeholder for the remote object and is available locally. It can be programmed to connect to an active instance of the remote object to ensure high availability.

Virtual proxy

Figure 9.4 exhibits a virtual proxy that acts as a surrogate to the original object. It is used to represent objects that interact with data or resources and consume time during their interaction with them. The virtual proxy object can be used to cache information and send it to the client. While the call to the original object returns, the updates can then be pushed to the client.

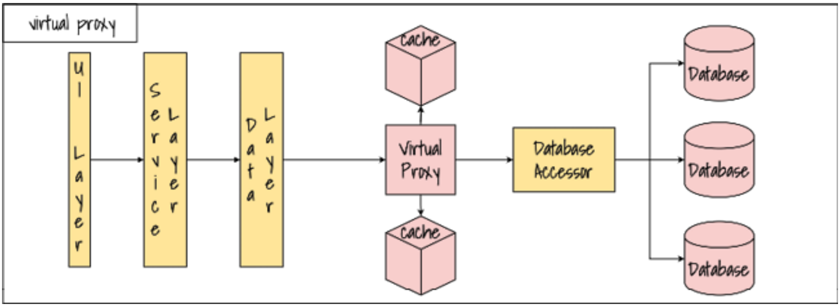

virtual proxy
UI Layer
Service Layer
Data Layer
Cache
Virtual Proxy
Cache
Database Accessor
Database
Database
Database

***Figure 9.4**: Virtual proxy*

Protection proxy

Figure 9.5 exhibits a protection proxy that guards access to another object could be used to perform access checks and improve the security for critical resources. It could connect to another service to confirm that the client has access to the requested object or the resource it guards. It then either provides the access or restricts it based on the response received from the service.

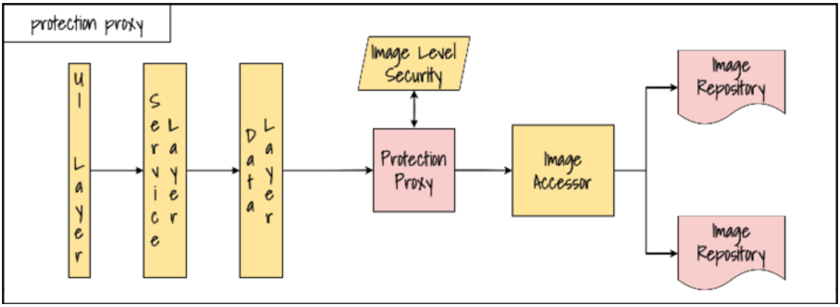

protection proxy
UI Layer
Service Layer
Data Layer
Image Level Security
Protection Proxy
Image Accessor
Image Repository
Image Repository

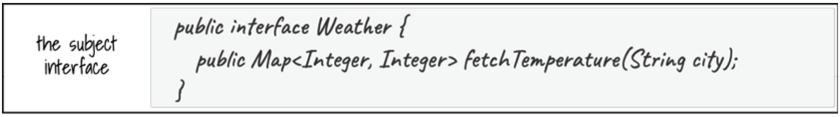
***Figure 9.5**: Protection proxy*

Building a proxy

The example that we will discuss in this chapter makes use of an interface that provides methods to access weather-related information, a concrete implementation of it, and a proxy that can replace it and provide additional functionality on top of it. We will learn how the proxy design pattern can be utilized to construct proxy objects that can represent existing object instances. Let us first observe the interface that sets up the common contract that both the concrete implementation and the proxy must follow for the design to work.

Interface

The interface is the focal point of the proxy design pattern. In *Figure 9.6*, the **Weather** interface provides the behavior that must be implemented by the concrete implementations as well as the proxies:



the subject interface

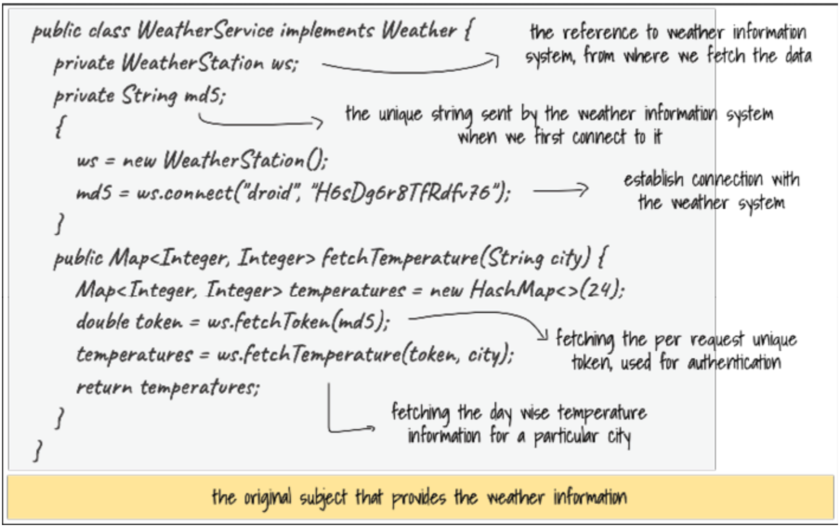
```
public interface Weather {
   public Map<Integer, Integer> fetchTemperature(String city);
}
```

Figure 9.6: *Interface Weather*

This interface provides us with a single method that can be used to fetch temperature values for the current day, for a given city.

Subject

In *Figure 9.7*, **WeatherService** is a concrete implementation of the **Weather** interface. It fetches the weather information whenever a request is received. When an instance of this service is instantiated, it establishes a connection with the weather station and reuses it to provide the desired functionality.



```
public class WeatherService implements Weather {
   private WeatherStation ws;
   private String md5;
   {
      ws = new WeatherStation();
      md5 = ws.connect("droid", "H6sDg6r8TfRdfv76");
   }
   public Map<Integer, Integer> fetchTemperature(String city) {
      Map<Integer, Integer> temperatures = new HashMap<>(24);
      double token = ws.fetchToken(md5);
      temperatures = ws.fetchTemperature(token, city);
      return temperatures;
   }
}
```

the reference to weather information system, from where we fetch the data

the unique string sent by the weather information system when we first connect to it

establish connection with the weather system

fetching the per request unique token, used for authentication

fetching the day wise temperature information for a particular city

the original subject that provides the weather information

Figure 9.7: *Weather service | Original object*

The implementation in *Figure 9.7* sends out a request to the weather station every single time. With an increase in the number of requests, the performance of this service might degrade, assuming that there are no limits on the number of requests that could be processed in a specific timeframe.

Proxy

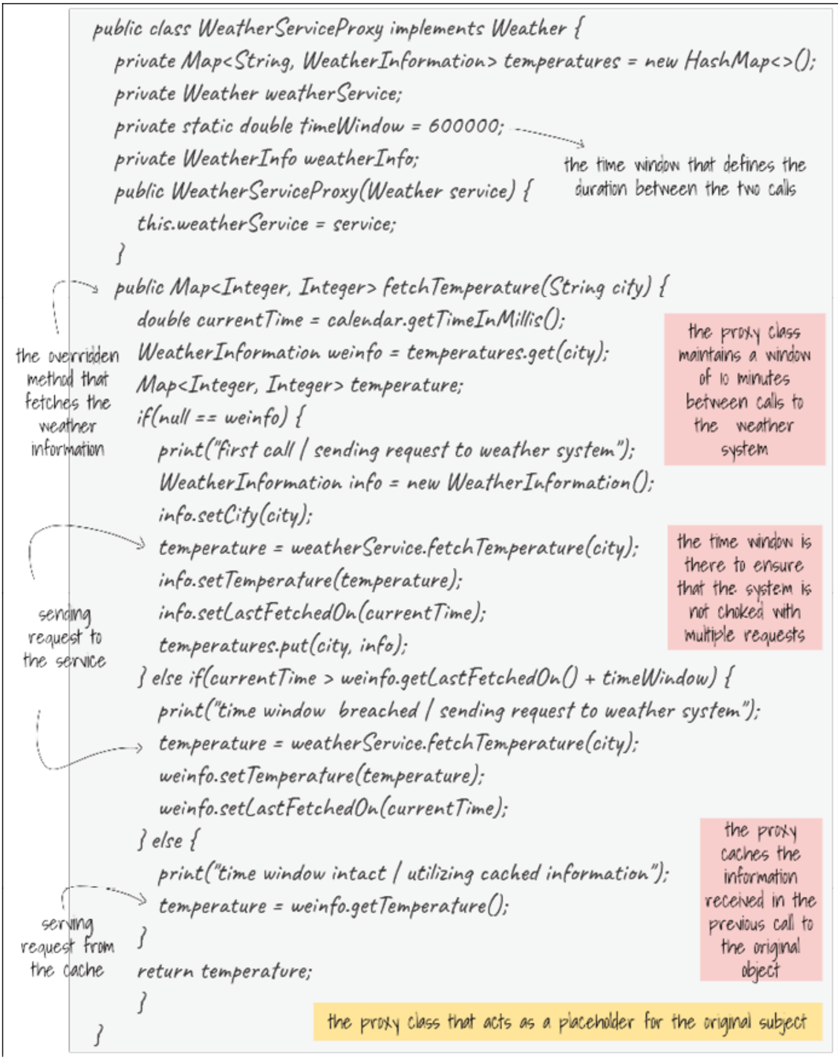
The proxy class implements the same interface as the subject or the original class and wraps an object of the same interface type. The proxy class imitates the structure of the subject class and, therefore, could easily replace it. Once the proxy is inserted into the call stack, it is hardly noticeable and has no impact on existing integrations.

The proxy class receives all the requests, which were earlier received by the subject, works on it, forwards the request to the original subject, receives the response, works on it again if required, and finally sends the response to the client.

In *Figure 9.8*, the proxy class implements the **Weather** interface and has a reference to the original subject type as well. The proxy class is structured in a manner such that it can act as a proxy to whichever object is passed to it, be it a concrete implementation of that type or another proxy that implements the same interface.

Our implementation of the proxy class adds a layer of caching on top of the wrapped subject's implementation. It maintains a local cache that is updated from time to time, as per the defined time window. Whenever a request lands on the proxy object, it first checks whether it could be served from the local cache. If the local cache has the required information and the time window has not lapsed, the result is served from the cache. Only when either the cache does not have the information or the time window for that information has lapsed, it makes a request to the original subject, fetches the response, and updates the local cache.

The proxy object, with its implementation of a cache, limits the number of requests that end up at the original subject and increases the throughput. It also reduces latency and the chances of performance degradation due to multiple requests choking the weather service.



```
public class WeatherServiceProxy implements Weather {
    private Map<String, WeatherInformation> temperatures = new HashMap<>();
    private Weather weatherService;
    private static double timeWindow = 600000;
    private WeatherInfo weatherInfo;
    public WeatherServiceProxy(Weather service) {
        this.weatherService = service;
    }
    public Map<Integer, Integer> fetchTemperature(String city) {
        double currentTime = calendar.getTimeInMillis();
        WeatherInformation weinfo = temperatures.get(city);
        Map<Integer, Integer> temperature;
        if(null == weinfo) {
            print("first call / sending request to weather system");
            WeatherInformation info = new WeatherInformation();
            info.setCity(city);
            temperature = weatherService.fetchTemperature(city);
            info.setTemperature(temperature);
            info.setLastFetchedOn(currentTime);
            temperatures.put(city, info);
        } else if(currentTime > weinfo.getLastFetchedOn() + timeWindow) {
            print("time window breached / sending request to weather system");
            temperature = weatherService.fetchTemperature(city);
            weinfo.setTemperature(temperature);
            weinfo.setLastFetchedOn(currentTime);
        } else {
            print("time window intact / utilizing cached information");
            temperature = weinfo.getTemperature();
        }
        return temperature;
        }
}
```

the time window that defines the duration between the two calls

the overridden method that fetches the weather information

the proxy class maintains a window of 10 minutes between calls to the weather system

sending request to the service

the time window is there to ensure that the system is not choked with multiple requests

the proxy caches the information received in the previous call to the original object

serving request from the cache

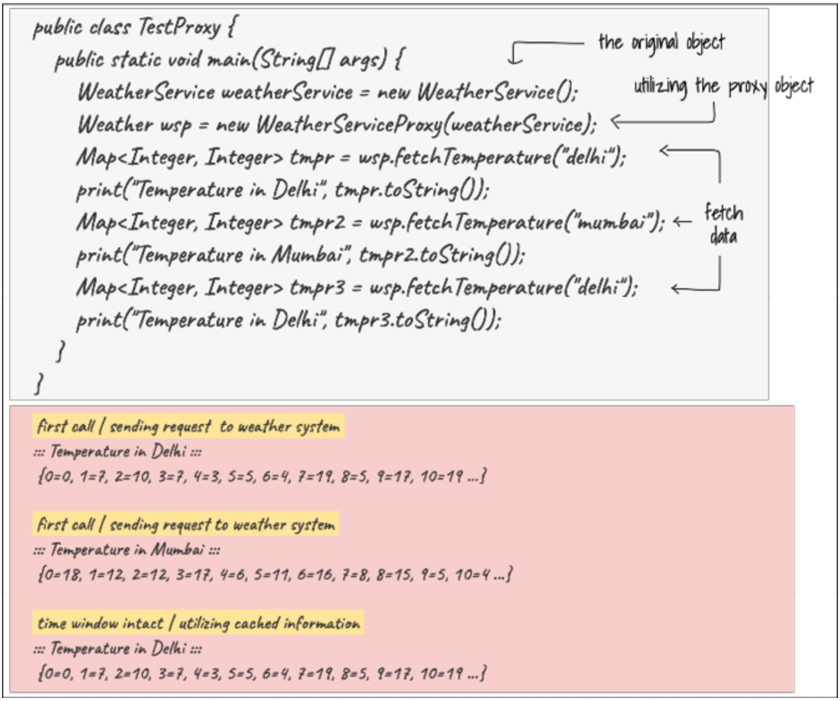
the proxy class that acts as a placeholder for the original subject

***Figure 9.8**: Weather service proxy*

Testing the proxy

We have written a test class to verify the working of the proxy design pattern. Let us execute the same.

In *Figure 9.9*, we start by instantiating an original subject and a proxy object, passing the subject to it. Then we make three calls to the proxy instead of the subject. The first two calls pass a different value for the method argument, but the third call passes an already used value. Response received from the proxy instance verifies that the third call was served using the cache and that no request was sent to the original object.



```
public class TestProxy {
  public static void main(String[] args) {
    WeatherService weatherService = new WeatherService();
    Weather wsp = new WeatherServiceProxy(weatherService);
    Map<Integer, Integer> tmpr = wsp.fetchTemperature("delhi");
    print("Temperature in Delhi", tmpr.toString());
    Map<Integer, Integer> tmpr2 = wsp.fetchTemperature("mumbai");
    print("Temperature in Mumbai", tmpr2.toString());
    Map<Integer, Integer> tmpr3 = wsp.fetchTemperature("delhi");
    print("Temperature in Delhi", tmpr3.toString());
  }
}
```

```
first call / sending request  to weather system
::: Temperature in Delhi :::
 {0=0, 1=7, 2=10, 3=7, 4=3, 5=5, 6=4, 7=19, 8=5, 9=17, 10=19 ...}

first call / sending request to weather system
::: Temperature in Mumbai :::
 {0=18, 1=12, 2=12, 3=17, 4=6, 5=11, 6=16, 7=8, 8=15, 9=5, 10=4 ...}

time window intact / utilizing cached information
::: Temperature in Delhi :::
 {0=0, 1=7, 2=10, 3=7, 4=3, 5=5, 6=4, 7=19, 8=5, 9=17, 10=19 ...}
```

***Figure 9.9**: Test proxy*

Summary

The proxy design pattern, as illustrated in the following figure, provides a way to create proxies that can be used to represent other objects. A proxy has the same structure as the original subject, implements the same interface, and wraps a subject instance.

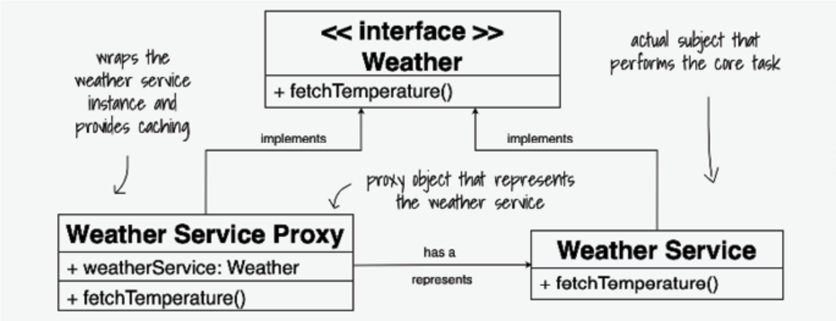

wraps the weather service instance and provides caching
<< interface >>
Weather
+ fetchTemperature()
actual subject that performs the core task
implements
implements
proxy object that represents the weather service
Weather Service Proxy
+ weatherService: Weather
+ fetchTemperature()
has a
represents
Weather Service
+ fetchTemperature()

Figure 9.10*: Proxy design pattern defined*

Advantages and drawbacks

It is time to make note of some of the benefits and drawbacks of the proxy design pattern. Let us start with the benefits first followed by the drawbacks:

- Access to an object can be controlled without changing the way the object is being accessed.
- Multiple proxies can be chained together in a sequence, where each proxy is responsible for a specific task.
- A proxy can imitate a real object when it is not ready to take requests.
- Proxies can be attached to the original object at runtime.
- Promotes separation of concern.

The drawbacks of the proxy design pattern are as follows:

- If not designed properly, the proxies can end up calling themselves in a loop.
- Proxies can lead to bottlenecks when the incoming requests are overwhelmingly large in number.

Usage

The following list outlines some of the scenarios that are relevant to constructing a proxy:

- Guarding or controlling access to an object.
- Caching or logging information every time an object is

accessed.

- Managing access to a remote object.
- Massaging input before it reaches the original object.
- Performing cleanup operations.

Conclusion

The proxy design pattern is a widely used structural design pattern that can be utilized to create proxies that represent existing objects without altering their structure. There can be three types of proxy objects, namely remote proxy, virtual proxy, and protection proxy. The proxies can be chained together to represent either the original subject or another proxy object.

In the next chapter, we will discuss another structural design pattern known as the facade design pattern and will learn how to simplify complex systems by introducing easy-to-use interfaces that hide the complex internals of a system.

Questions

In this section, we will sharpen our brains by indulging in some fun brain activities. Let us get started.

Find the odd one out

1. Which of the following adjectives cannot be used for a proxy:
    a. Placeholder
    b. Substitute
    c. Implementation
    d. Representative
2. Which of the following proxies are not discussed in this chapter:
    a. Virtual
    b. Conversion
    c. Remote

    - d. Protection
3. Which of the following does not manage another object:
    - a. Filter
    - b. Proxy
    - c. Guardian
    - d. Adapter
4. Proxy design pattern:
    - a. Creates proxies to represent existing objects
    - b. Requires modifications in the existing code
    - c. Requires changes in existing integration
    - d. Is a structural design pattern
5. Proxy design pattern is broadly categorized as:
    - a. Virtual proxy
    - b. Remote proxy
    - c. Converter proxy
    - d. Protection proxy
6. The proxy design pattern can be used to:
    - a. Perform lazy initialization
    - b. Replace the original object
    - c. Cache information
    - d. Safeguard access to the original object

Select two correct answers

1. Proxy design pattern:
    - a. Replaces existing objects
    - b. Requires modifications in the existing code
    - c. Requires a base interface to implement
    - d. Can safeguard access to objects
2. The proxy design pattern is most useful for representing:
    - a. Objects with high latency

    b. Model objects
    c. Other proxies
    d. Objects that need to be secured
3. What goes together?
    a. Security, access control, protection proxy
    b. Chain of proxies, high coupling, remote proxy
    c. Sub-classing, inheritance, proxy design pattern
    d. Caching, database, direct access

Answers

This section provides the answers to the questions given.

Find the odd one out

1. Implementation
2. Conversion
3. Adapter
4. Requires modifications in existing code
5. Converter proxy
6. Replace the original object

Select two correct answers

1. Requires a base interface to implement, Can safeguard access to objects.
2. Objects with high latency, objects that need to be secured.
3. Security, access control, protection proxy, and Chain of proxies, high coupling, remote proxy

Join our Discord space

Join our Discord workspace for latest updates, offers, tech happenings around the world, new releases, and sessions with the authors:

https://discord.bpbonline.com

CHAPTER 10

Simplifying the Complexity

Introduction

Using interfaces to expose services is one of the ideal design solutions for a software application. An interface describes a contract between a service and its users and provides methods to facilitate interaction. An application can have multiple interfaces; each is responsible for providing the required abstraction to allow classes to interact with each other and ensuring that they remain loosely coupled.

Integration that involves multiple interfaces could easily become complex if the interfaces are not well-defined or if the desired behavior requires interaction with multiple interfaces. The order of execution is important when you are interacting with a system that involves multiple sub-systems, which could add to the complexity of integrating with the overall system.

Figure 10.1 exhibits distinct interfaces and their implementations. It symbolizes the complexity and the effort required to integrate with each of them:

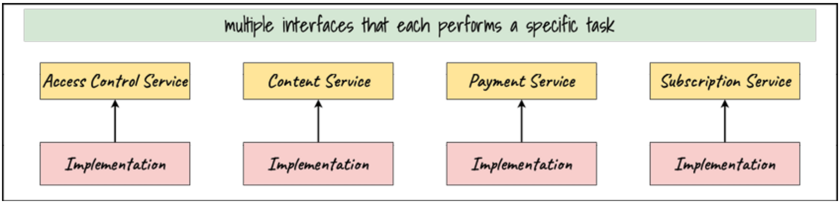

multiple interfaces that each performs a specific task
Access Control Service
Content Service
Payment Service
Subscription Service
Implementation
Implementation
Implementation
Implementation

Figure 10.1*: Individual interfaces*

To make a complex system easier to use and integrate with, a unified interface could be provided that minimizes the dependency of the clients on the multiple interfaces exposed by the system and reduces the complexity by exposing methods that present an abstract view of the system to the outside world.

In this chapter, we will learn about the facade design pattern and how it can be utilized to ease the integration between a system and its users.

Structure

This chapter covers the following topics:

- Facade
- Facade design pattern
- Building a facade
- Summary
- Advantages and drawbacks
- Usage

Objectives

By the end of this chapter, the readers will be able to understand the complexity of integrating two systems together and how abstraction can help to simplify it. They will learn about the façade design pattern and how it can be utilized to provide a simple and unified interface that helps to reduce the complexity experienced during integrations.

The chapter will also walk them through the implementation details of a facade design pattern, the benefits it provides, and a few drawbacks, along with some of the places where this design pattern can be used.

Facade

A facade is an abstraction that hides the underlying complexity of a system and presents a simpler view of it to the outside world. In literal terms, a facade is the front wall of a building that the on-goers see from the outside, unaware of the complex structure housed within it.

In computing, a facade is an interface that provides a simple and unified view of a system by hiding the complexity of its various sub-systems that interact and cooperate to present a holistic view of the overall system. In other words, a facade groups together multiple functionalities that are usually performed in some order and presents them in the form of easy-to-use methods. These methods can be used to integrate with the system without delving into the underlying complexity.

In *Figure 10.2*, Service X and Service Y are interacting with various interfaces available in a system. A facade can help here by providing a simpler point of interaction:

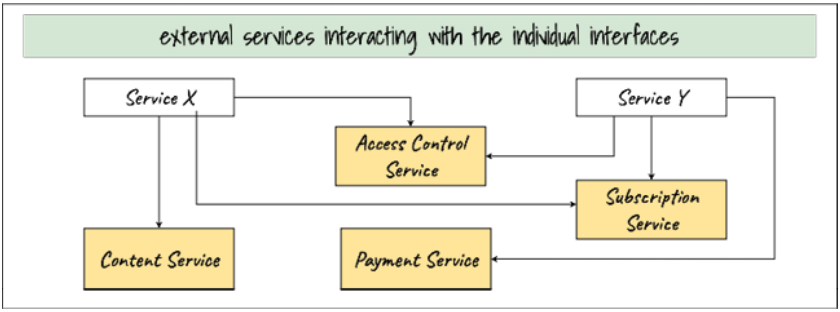

external services interacting with the individual interfaces
Service X
Service Y
Access Control Service
Subscription Service
Content Service
Payment Service

***Figure 10.2**: Service and interface interaction*

Facade design pattern

The facade design pattern is a structural design pattern that allows for the development of a facade. It provides a simple and extensible way to interact with a system that consists of various subsystems. *Figure 10.3* exhibits the usefulness of a facade design pattern for external services while interacting with a system that exposes multiple interfaces:

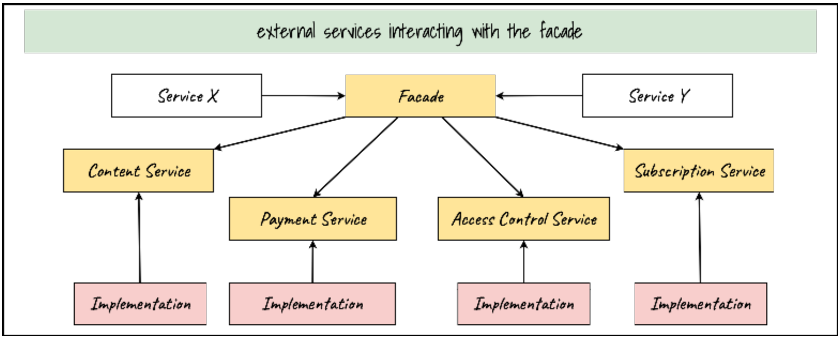

external services interacting with the facade
Service X
Facade
Service Y
Content Service
Payment Service
Access Control Service
Subscription Service
Implementation
Implementation
Implementation
Implementation

Figure 10.3: *Introducing facade*

A facade simplifies the overall integration process by providing a single interface that internally delegates the calls to sub-systems and relieves its clients from the complexity of integrating with most of them. Integrating with such a large system would otherwise require multiple integrations, involving numerous interfaces, as shown in the following figure:

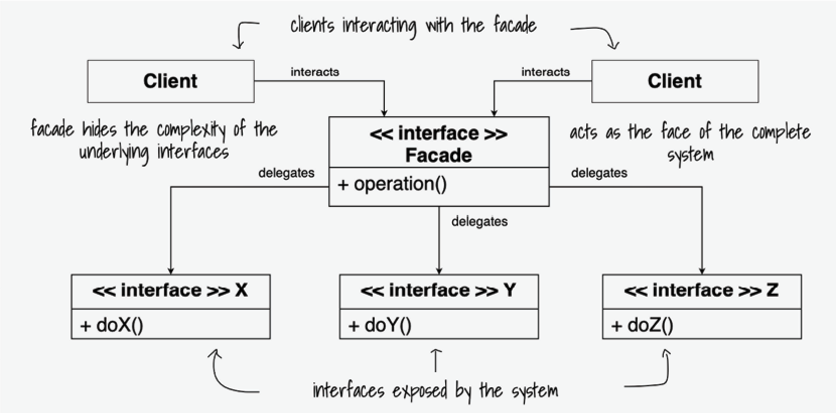

clients interacting with the facade
Client
interacts
interacts
Client
facade hides the complexity of the underlying interfaces
<< interface >>
Facade
+ operation()
acts as the face of the complete system
delegates
delegates
delegates
<< interface >> X
+ doX()
<< interface >> Y
+ doY()
<< interface >> Z
+ doZ()
interfaces exposed by the system

Figure 10.4: *The facade design pattern*

As shown in *Figure 10.4*, a facade design pattern allows for the creation of an interface of interfaces that provides simplified methods to the clients and interacts with the existing sub-systems itself, all while effectively hiding the complexity exhibited by those sub-systems. This reduces the burden on the external systems to integrate with multiple interfaces. It successfully hides internal complexity behind a unified interface that appears simple to the clients.

Here is how it works:

- Instantiate an object of the facade type that provides the simplified method definitions.
- The facade instance acts as a multiplexer and delegates the calls to multiple interfaces, gathers their output, and processes it if required before finally sending a response to the client.

In the next section, we will demonstrate the working and usage of the facade design pattern.

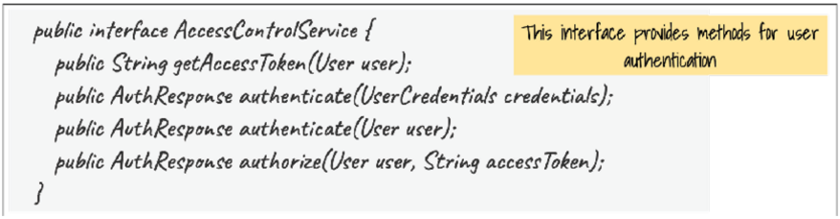
Building a facade

The example that we will walk you through in this chapter makes use of multiple interfaces that provide various methods to access the online services of an over-the-top platform. We will learn about implementing a facade and how it benefits the clients by providing simple and ready-to-use methods. Let us first look at the interfaces that provide various functionalities required to access the over-the-top platform, and later, we will discuss the facade implementation.

All for one

A system usually has multiple interfaces that are accessed by its clients to avail the services provided by it. Many of those interfaces handle only a part of the task that the client intends to perform, requiring the client to make multiple calls to the system. The client must take care of multiple integrations and handle any changes that are made to those interfaces.

The system we have chosen to present as a use case for the facade design pattern also exposes multiple interfaces that the client can integrate to achieve the desired results. Assuming the client only must integrate with the existing implementations, it still must ensure the order in which the calls are made to these interfaces and handle the exception or error scenarios, as shown in the following figure:

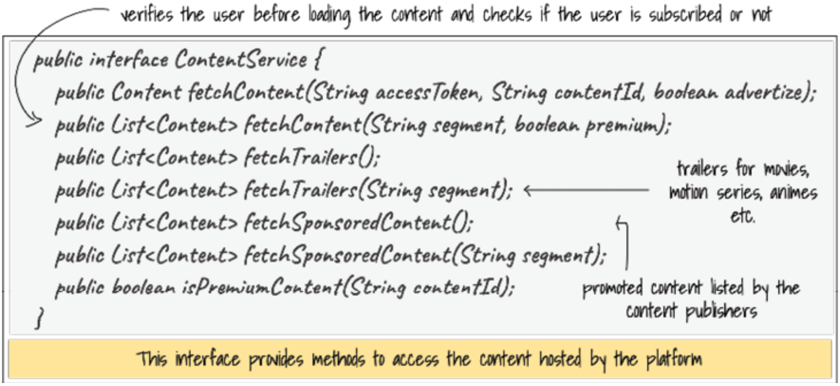


```
public interface AccessControlService {
    public String getAccessToken(User user);
    public AuthResponse authenticate(UserCredentials credentials);
    public AuthResponse authenticate(User user);
    public AuthResponse authorize(User user, String accessToken);
}
```

This interface provides methods for user authentication

***Figure 10.5**: An interface for access control*

In *Figure 10.5*, the interface provides multiple methods to control user access. It exposes methods to authenticate and authorize the user and fetch a unique token that is used to access the content from the platform. In *Figure 10.6*, the interface lists the methods available to access the digital content available on the platform:



verifies the user before loading the content and checks if the user is subscribed or not

```
public interface ContentService {
    public Content fetchContent(String accessToken, String contentId, boolean advertize);
    public List<Content> fetchContent(String segment, boolean premium);
    public List<Content> fetchTrailers();
    public List<Content> fetchTrailers(String segment);
    public List<Content> fetchSponsoredContent();
    public List<Content> fetchSponsoredContent(String segment);
    public boolean isPremiumContent(String contentId);
}
```

trailers for movies, motion series, animes etc.

promoted content listed by the content publishers

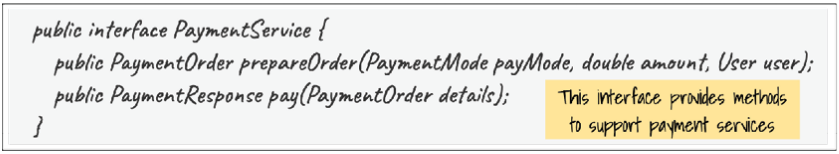
This interface provides methods to access the content hosted by the platform

***Figure 10.6**: An interface to access digital content*

It provides multiple methods to fetch the digital content, for example, trailers available for the movies, motion series, anime, etc., available on the platform. Not just that, it also has methods to provide promoted or sponsored content.

We will make use of four interfaces for our example, and we have already discussed two of them that are most relevant to the functionality our example demonstrates. The other two interfaces are there to support content monetization; one is the payment interface, and the other is responsible for subscribing and unsubscribing users to the platform.

In *Figure 10.7*, the interface provides methods that can be utilized by the client to initiate and close payment orders related to user subscription. The first method prepares a payment order that is added to the user's cart during checkout, and the second method helps to establish a connection with payment providers, complete the payment, and subscribe the user to the platform.

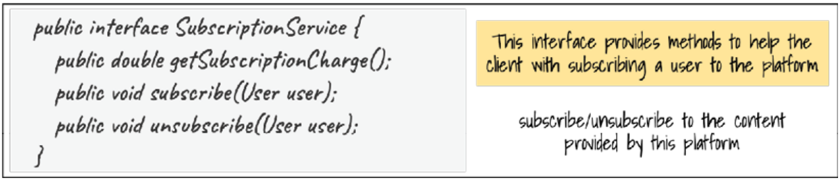


```
public interface PaymentService {
   public PaymentOrder prepareOrder(PaymentMode payMode, double amount, User user);
   public PaymentResponse pay(PaymentOrder details);
}
```

This interface provides methods to support payment services

***Figure 10.7**: An interface to support payment service*

In *Figure 10.8*, the interface provides methods that help with subscribing and unsubscribing a user to the platform, as per their consent and the confirmation of payment in case of subscription:



```
public interface SubscriptionService {
   public double getSubscriptionCharge();
   public void subscribe(User user);
   public void unsubscribe(User user);
}
```

This interface provides methods to help the client with subscribing a user to the platform

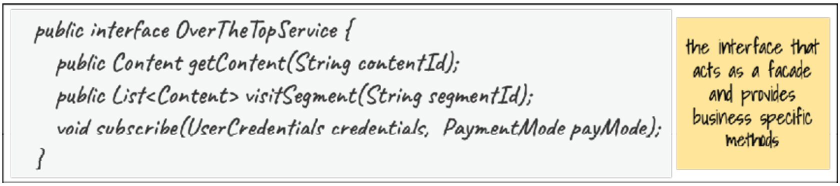
subscribe/unsubscribe to the content provided by this platform

***Figure 10.8**: An interface to support subscription and unsubscription*

All these interfaces are good enough to be directly utilized by the client, but there are too many to integrate with. It would be helpful to have fewer interfaces to integrate with, as it would reduce the cost of integrating with interfaces and testing to confirm their correct functioning. Not just that, the developers also have to ensure to call these methods in the correct order, something that can be simplified if the ordering of these calls can be cumulated, and an exoskeleton is presented to the client instead of individual interfaces. All this can be achieved by utilizing the facade design pattern and exposing methods that relate more to the business flow rather than the technical side of it. Let us now dive into creating another interface that acts as a facade and makes our lives easier.

One for all

A facade is like an interface of interfaces. The facade design pattern provides a unified interface that is easier to understand and integrate with. It hides the complexity exhibited by the many interfaces that the facade implementation interacts with, all while providing a simpler interface to the clients. In *Figure 10.9*, the interface provides methods that are more in line with the business use cases and make more sense to the clients:



```
public interface OverTheTopService {
   public Content getContent(String contentId);
   public List<Content> visitSegment(String segmentId);
   void subscribe(UserCredentials credentials, PaymentMode payMode);
}
```

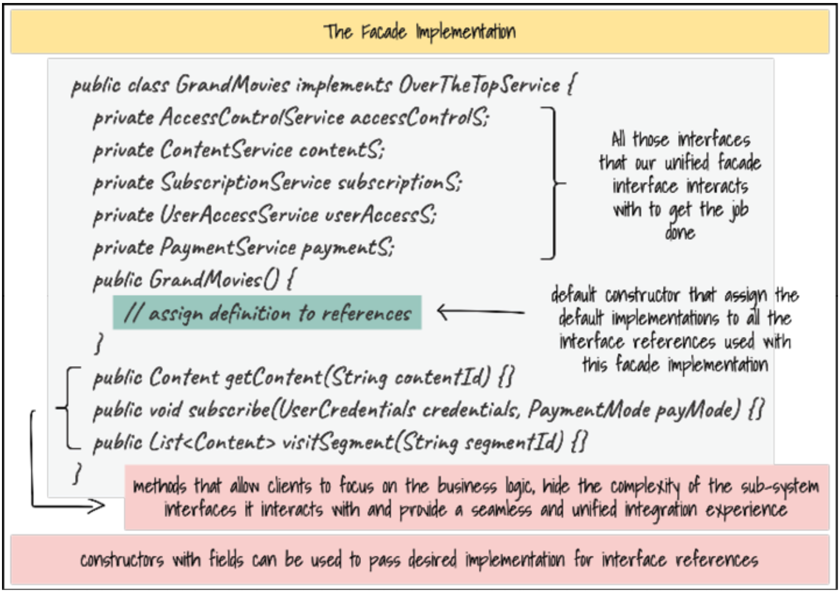
the interface that acts as a facade and provides business specific methods

***Figure 10.9**: The facade interface*

The clients can integrate with one unified interface and achieve the desired output instead of integrating with all the other interfaces provided by the system. This relieves the client from multiple complex integrations and drives the focus towards business logic and its implementation.

Let us now look at one of the many possible implementations of the facade interface. Integrating with interfaces and not the implementations allows for dynamic behavior, based on the selected implementation, and opens the possibility to work with multiple approaches without changing the contract.

In *Figure 10.10*, the implementation class presents a way to interact with a set of interfaces to provide a simple and unified integration endpoint for the many clients of the system. If you have noticed, we have used a default constructor that assigns pre-selected implementations to the references of various interface types and provides a default behavior. To change this behavior, we can use constructors with fields and pass the desired implementation that we want to use with the facade.



The Facade Implementation

```
public class GrandMovies implements OverTheTopService {
   private AccessControlService accessControlS;
   private ContentService contentS;
   private SubscriptionService subscriptionS;
   private UserAccessService userAccessS;
   private PaymentService paymentS;
   public GrandMovies() {
      // assign definition to references
   }
   public Content getContent(String contentId) {}
   public void subscribe(UserCredentials credentials, PaymentMode payMode) {}
   public List<Content> visitSegment(String segmentId) {}
}
```

All those interfaces that our unified facade interface interacts with to get the job done

default constructor that assign the default implementations to all the interface references used with this facade implementation

methods that allow clients to focus on the business logic, hide the complexity of the sub-system interfaces it interacts with and provide a seamless and unified integration experience

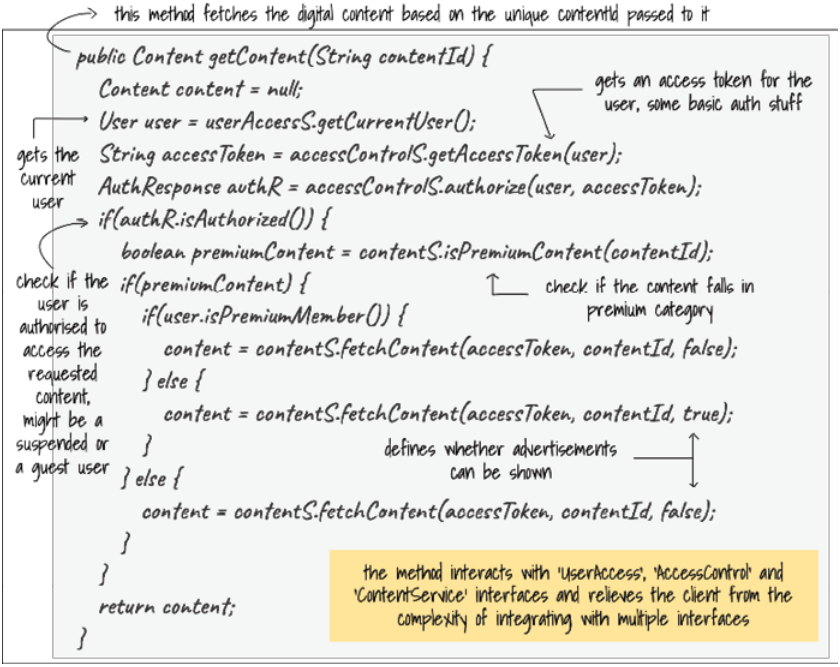
constructors with fields can be used to pass desired implementation for interface references

***Figure 10.10**: A facade implementation*

Let us now look at the method definitions that form the logic that our facade implementation provides to simplify the integration process. We will go through the methods one by one in the order in which they were listed in *Figure 10.10*.

These implementations are only for understanding the concepts and might not be relevant for actual business logic. They are also not suitable for production-level code, actually, far from it. You are advised to exercise caution and to only use it for learning purposes.

In *Figure 10.11*, the method definition can be seen interacting with multiple interfaces. It is responsible for fetching the digital content based on the unique content ID passed to it. The method does a lot more than just fetching the content, though; it takes care of user authorization, content validation, and accessing the right content. It also ensures that the calls to all the interconnecting interfaces are made in the correct order.



this method fetches the digital content based on the unique contentId passed to it

```
public Content getContent(String contentId) {
  Content content = null;
  User user = userAccessS.getCurrentUser();
  String accessToken = accessControlS.getAccessToken(user);
  AuthResponse authR = accessControlS.authorize(user, accessToken);
  if(authR.isAuthorized()) {
    boolean premiumContent = contentS.isPremiumContent(contentId);
    if(premiumContent) {
      if(user.isPremiumMember()) {
        content = contentS.fetchContent(accessToken, contentId, false);
      } else {
        content = contentS.fetchContent(accessToken, contentId, true);
      }
    } else {
      content = contentS.fetchContent(accessToken, contentId, false);
    }
  }
  return content;
}
```

gets an access token for the user, some basic auth stuff

gets the current user

check if the user is authorised to access the requested content, might be a suspended or a guest user

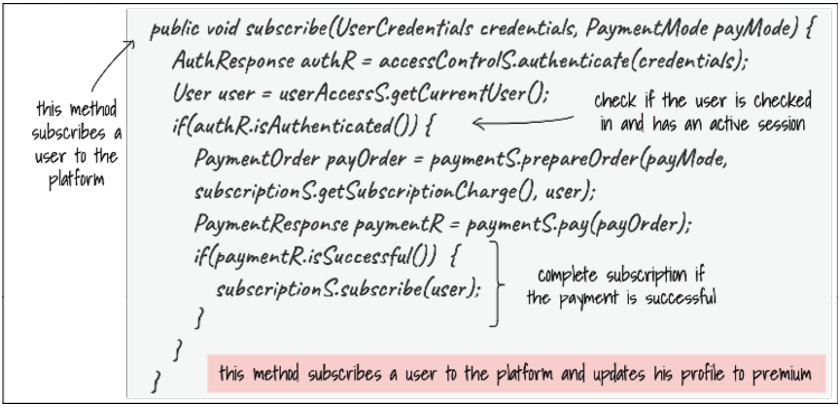
check if the content falls in premium category

defines whether advertisements can be shown

the method interacts with 'UserAccess', 'AccessControl' and 'ContentService' interfaces and relieves the client from the complexity of integrating with multiple interfaces

Figure 10.11: *Implementing facade I*

Figures 10.12 and *10.13* present us with the definition of the other two methods of our facade implementation. The method described in *Figure 10.12* is responsible for subscribing a user to the platform and promoting their profile to a premium user. It takes care of user authentication, creating a payment order, and finally completing it.



```
public void subscribe(UserCredentials credentials, PaymentMode payMode) {
  AuthResponse authR = accessControlS.authenticate(credentials);
  User user = userAccessS.getCurrentUser();
  if(authR.isAuthenticated()) {
    PaymentOrder payOrder = paymentS.prepareOrder(payMode,
    subscriptionS.getSubscriptionCharge(), user);
    PaymentResponse paymentR = paymentS.pay(payOrder);
    if(paymentR.isSuccessful()) {
      subscriptionS.subscribe(user);
    }
  }
}
```

this method subscribes a user to the platform

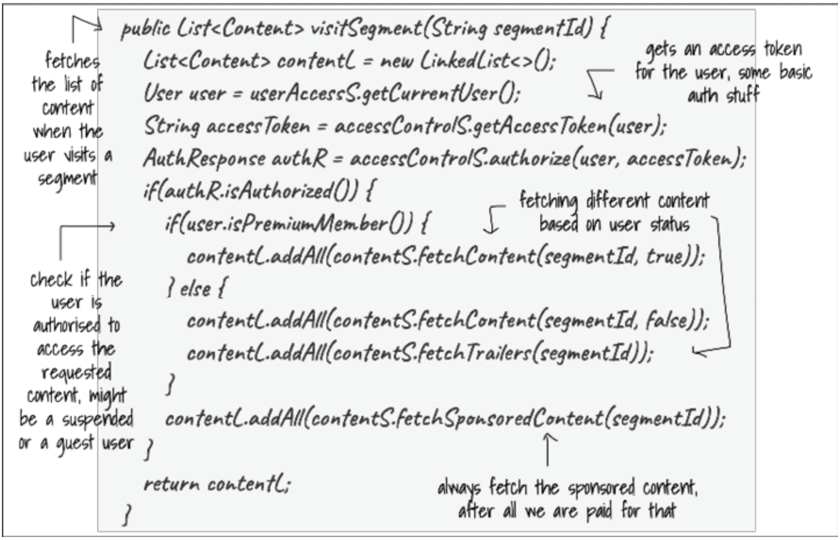
check if the user is checked in and has an active session

complete subscription if the payment is successful

this method subscribes a user to the platform and updates his profile to premium

Figure 10.12: *Implementing facade II*

The second method, the one described in *Figure 10.13*, fetches all the available content whenever a user visits a particular segment. Both these methods interact with multiple interfaces, like the method definition in *Figure 10.11*:



```
public List<Content> visitSegment(String segmentId) {
    List<Content> contentL = new LinkedList<>();
    User user = userAccessS.getCurrentUser();
    String accessToken = accessControlS.getAccessToken(user);
    AuthResponse authR = accessControlS.authorize(user, accessToken);
    if(authR.isAuthorized()) {
        if(user.isPremiumMember()) {
            contentL.addAll(contentS.fetchContent(segmentId, true));
        } else {
            contentL.addAll(contentS.fetchContent(segmentId, false));
            contentL.addAll(contentS.fetchTrailers(segmentId));
        }
        contentL.addAll(contentS.fetchSponsoredContent(segmentId));
    }
    return contentL;
}
```

fetches the list of content when the user visits a segment
gets an access token for the user, some basic auth stuff
fetching different content based on user status
check if the user is authorised to access the requested content, might be a suspended or a guest user
always fetch the sponsored content, after all we are paid for that

Figure 10.13*: Implementing facade III*

If you look closely, you will find that these methods are doing the exact same thing a client would do to integrate with the multiple interfaces exposed by the system. They will end up writing something very similar to a facade. The obvious difference would be that the client will have to work through the complexity of integrating with those interfaces.

By using a facade, the whole integration process can be largely simplified for the clients of the system. The clients no longer need to integrate with multiple interfaces exposed by the system or to take care of maintaining the integrations when changes are made to those interfaces. Overall, it greatly reduces the complexity and helps them focus on the core business logic.

In other words, the facade design pattern provides a higher-level interface that makes the subsystem easier to use and interact with.

Summary

The facade design pattern allows the construction of a simplified and unified interface on top of the existing interfaces. It hides the complexity of the sub-systems from the clients and provides a simplified view of the overall system. It ensures that the client is relieved from handling multiple integrations by themselves and exposes a single interface that the clients can rely on.

The facade design pattern is most useful for large systems that involve multiple interfaces. By limiting the number of integration points with the clients, it also helps to reduce the gaps in how they interact with the system and subsequently decreases the number of integration issues. *Figure 10.14* exhibits a class diagram that represents the facade design pattern. It makes use of the interfaces and classes from the example in this chapter:

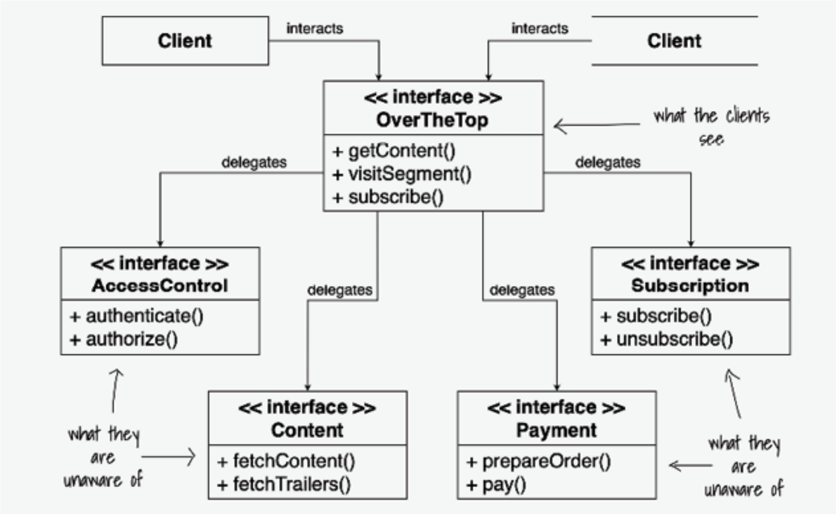

Client
interacts
interacts
Client
<< interface >>
OverTheTop
+ getContent()
+ visitSegment()
+ subscribe()
what the clients see
delegates
delegates
<< interface >>
AccessControl
+ authenticate()
+ authorize()
delegates
delegates
<< interface >>
Subscription
+ subscribe()
+ unsubscribe()
what they are unaware of
<< interface >>
Content
+ fetchContent()
+ fetchTrailers()
<< interface >>
Payment
+ prepareOrder()
+ pay()
what they are unaware of

***Figure 10.14**: Implementing facade IV*

Advantages and drawbacks

It is time to make note of some of the benefits and drawbacks of the facade design pattern. Let us start with the benefits first and then the drawbacks:

- It provides a simplified and unified interface to integrate with the system.
- It hides the complexity of the system from the outside world.

- It helps the clients to focus on the core business logic.
- It promotes loose coupling between the clients and the various sub-systems it interacts with.

There are no known drawbacks of the facade pattern. Though during its development, it should be made to interact with the interfaces and not directly with any of the implementations. This helps in switching to different implementations if desired, rather than sticking to a single implementation.

Usage

The following list outlines some of the scenarios that are relevant for building a facade:

- When there is a need to provide a single, unified interface for a complex, multi-layered system.
- When the dependencies between a system and clients are to be reduced.
- When there is a need to reduce the impact of future changes in the system on the clients.

Conclusion

The facade design pattern is a widely used structural design pattern that can be utilized to create facades that help to improve the integration process by providing methods that are more in line with the client's business. It hides the complexity of the system and simplifies the complex integration process.

In the next chapter, we will learn about the bridge design pattern and how it can be used to split large classes into smaller ones and develop them independently while reducing the maintenance effort.

Questions

In this section, we will sharpen our brains by indulging in some fun brain activities. Let us get started.

Find the odd one out

1. Facade, simplified, unified, complex
2. Reduces complexity, simplifies integration, decreases performance, and provides ready to use methods
3. Payment, subscription, refund, content
4. Facade design pattern:
    a. Provides a simplified interface
    b. Is a structural design pattern
    c. Allows for creation of multiple small interfaces
    d. Reduces integration complexity
5. Facade design pattern is useful when:
    a. Complexity of integration is to be reduced
    b. Clients perform integration
    c. Dependencies are to be reduced
    d. The impact of future changes should be limited
6. The facade design pattern:
    a. Can be used to develop more than one facade
    b. Limits the clients to use only facades
    c. Can interact with any number of internal sub-systems
    d. Relieves the client from major maintenance overload

Select two correct answers

1. Facade design pattern:
    a. Hides complexity from clients
    b. Allows clients to interact with any number on interfaces
    c. Provides a simplified interface to the clients
    d. Can be private
2. Facade design pattern can be used to reduce:
    a. Complexity
    b. Interactions
    c. Exceptions
    d. Methods exposed to the client

3. What goes together?
    a. Integration, system, client
    b. Reduce complexity, facade, simpler integration
    c. Increase complexity, unified interface, facade
    d. Facade, behavioral, multiple integrations

Answers

This section provides the answers to the given questions.

Find the odd one out

1. Complex
2. Decreases performance
3. Content
4. Allows for the creation of multiple small interfaces
5. Dependencies are to be reduced
6. Limits the clients to use only facades

Select two correct answers

1. Hides complexity from clients, provides a simplified interface to the clients.
2. Complexity, methods exposed to the client.
3. Integration, system, client, and reduce complexity, facade, simpler integration.

CHAPTER 11

Bridge

Introduction

While writing software, we often rely on abstraction and polymorphism to choose a course of action. Interfaces and abstract classes allow us to build abstraction layers that the clients interact with, and polymorphism ensures that the right implementation is called to get something done, as shown in the following figure:

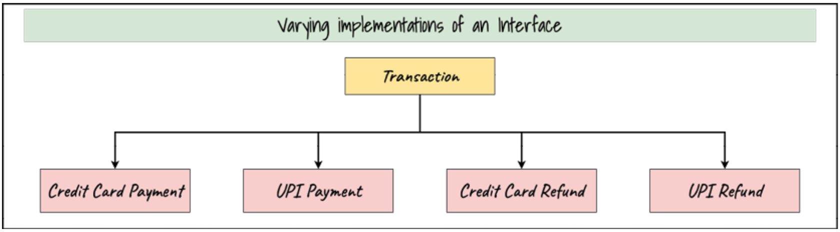

Varying implementations of an Interface
Transaction
Credit Card Payment
UPI Payment
Credit Card Refund
UPI Refund

***Figure 11.1**: Implementations of an interface*

Many times, the concrete implementations perform redundant tasks that are common across these implementations, but are nevertheless an important part of the entire logical implementation. For example, in *Figure 11.1*, the concrete implementations of the abstract class Transaction perform payment and refund operations using credit card and **Unified Payments Interface** (**UPI**) payment modes. Introduction of a new payment mode requires two new classes to ensure that both the payment and refund operations can be performed, while only the payment mode is being changed and not the operation.

Such a design not only increases redundancy but also the effort required to maintain it. Consider a small change required in the payment flow, and the necessary code changes required in each of the relevant implementations to make it happen. This architecture can be improved by separating the payment modes and operations performed on them.

Structure

This chapter covers the following topics:

- Bridge design pattern
- Implementing the bridge pattern
- Summary
- Advantages and drawbacks
- Usage

Objectives

By the end of this chapter, the readers will learn about the bridge design pattern that allows us to split complex classes into separate hierarchies and develop them independently. These hierarchies are often termed abstraction and implementation and are related via composition.

After finishing this chapter, they will be able to understand the workings of the bridge design pattern and how it can be used to reduce complexity and improve code clarity.

Bridge design pattern

The bridge design pattern is a structural design pattern that allows splitting a complex class or a group of classes into separate hierarchies. By doing so, they help to achieve independent development and maintenance of the classes that constitute those hierarchies.

In *Figure 11.1*, the child classes not only perform a type of transaction, but they do so for a distinct payment mode. This leads to multiple child classes being added, one each for a combination of payment mode and payment type, every time a new payment type or payment mode is introduced. For example, if internet banking is added as a payment mode in the future, then two new classes will be required to process payment and refund transactions. And further, if a new payment type, say reversal is added, then a new class would have to be added for each payment mode, increasing the number of combinations each time a payment mode or a payment type is added.

By extending the classes in two or more dimensions, we end up increasing the number of combinations, causing redundancy and complexity in the application design. The bridge design pattern aims to solve this problem by using composition in place of inheritance. It works by moving one of the dimensions to a separate class hierarchy that can be managed in isolation.

In our case, these implementations can be broken down into two separate hierarchies: *Transaction* and *Transaction Mode*, as exhibited in *Figure 11.2*:

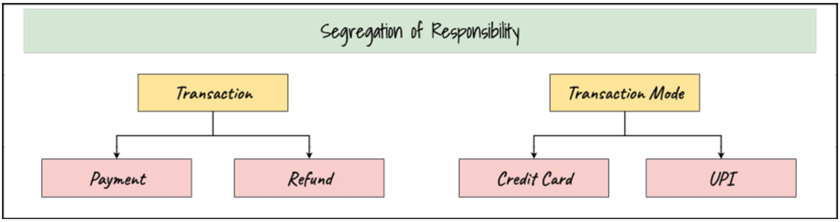


***Figure 11.2**: Segregation of responsibility*

Creating separate hierarchies allows us to divide the responsibility among classes that perform one specific kind of task, which in our case are classes for transaction types and transaction modes.

After the separation, the concrete implementations of the **Transaction** type are responsible for generic tasks like creating payment and refund orders, processing tax and billing information, whereas processing a transaction by integrating with a payment service provider is delegated to implementations of *TransactionMode*.

Figure 11.3 exhibits how these two hierarchies relate to each other using composition. It allows building distinct behaviors by integrating the two hierarchies together in different combinations:

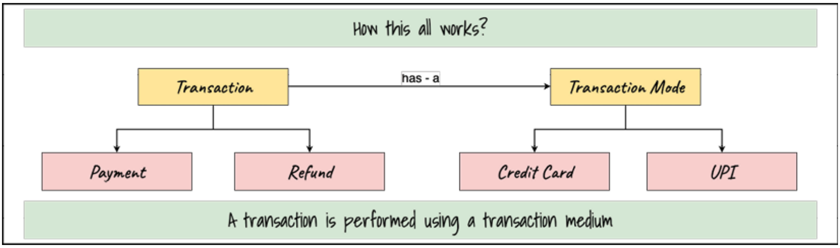

How this all works?
Transaction
has - a
Transaction Mode
Payment
Refund
Credit Card
UPI
A transaction is performed using a transaction medium

***Figure 11.3**: Abstraction and implementation*

Since the two hierarchies can now be developed independently, adding a new transaction type or transaction mode does not require the creation of multiple classes anymore. The newly added implementation can be seamlessly integrated with the existing classes to achieve new behaviors. *Figure 11.4* exhibits adding a new abstraction or implementation and its impact on the existing classes:

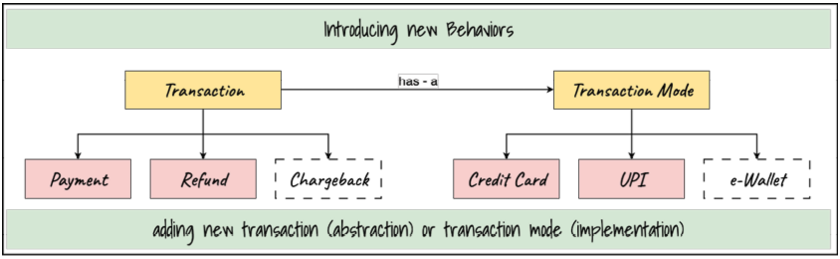

Introducing new Behaviors
Transaction
has - a
Transaction Mode
Payment
Refund
Chargeback
Credit Card
UPI
e-Wallet
adding new transaction (abstraction) or transaction mode (implementation)

***Figure 11.4**: Introducing new behaviors*

Figure 11.5 exhibits the architectural diagram of the bridge design pattern. It displays the interaction points between the two hierarchies and how the client interacts with the exposed abstraction layer. Polymorphism plays a vital role in the bridge design pattern. It not only allows the client to choose the correct *RefinedAbstraction* but also the *TransactionMode* that it interacts with to perform the desired operation.

TransactionMode, as an interface, creates a binding contract between itself and the *RefinedAbstraction* and imposes a mandatory requirement for all its implementations to provide a definition for all the methods that it exposes. When this is combined with polymorphism, it allows injecting distinct implementations programmatically, creating multiple behavioral possibilities using the same set of classes.

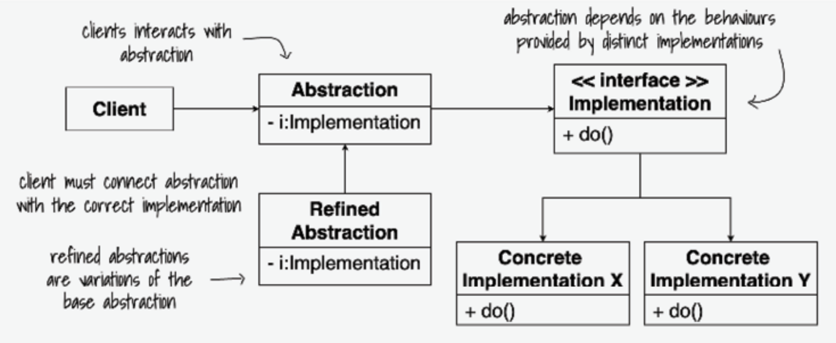

clients interacts with abstraction
abstraction depends on the behaviours provided by distinct implementations
Client
Abstraction
- i:Implementation
<< interface >>
Implementation
+ do()
client must connect abstraction with the correct implementation
Refined Abstraction
- i:Implementation
refined abstractions are variations of the base abstraction
Concrete Implementation X
+ do()
Concrete Implementation Y
+ do()

***Figure 11.5**: Bridge design pattern*

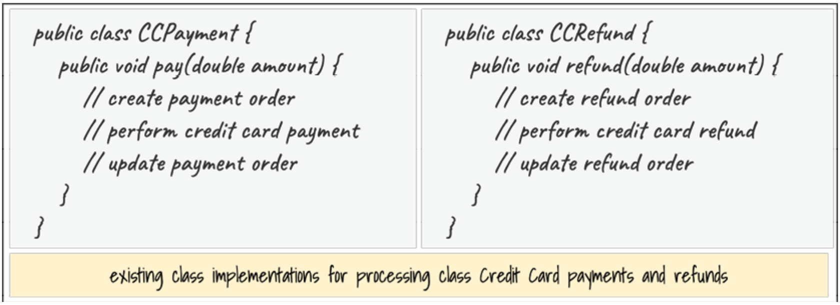
Implementing the bridge pattern

The example that we will walk you through in this chapter will explain the working of the bridge pattern effectively. We will use the bridge design pattern to improve upon an existing class hierarchy that processes money transfers using various transaction modes.

We will first look at the existing implementations of the **Payment** and **Refund** classes and understand the requirement for segregating their work into separate hierarchies. Then we will walk you through the new implementation using the bridge design pattern. We will learn about implementing the bridge design pattern and how it can be used to reduce the complexity of classes by splitting them into separate hierarchies and developing them individually.

Existing implementations

The existing implementation of payment and refund classes directly integrates with the payment service providers and requires fresh development whenever a new payment mode is introduced. This brings in tech debt in the form of code duplicity that can be hard to manage, especially when a change is required in one of the important flows, say order creation. With all that duplicate code, we may end up making changes in many classes. Do not forget the amount of testing required to test this change, which could have an impact on the delivery timelines. *Figure 11.6* exhibits existing implementations for processing payments and refunds:



```
public class CCPayment {
    public void pay(double amount) {
        // create payment order
        // perform credit card payment
        // update payment order
    }
}
```

```
public class CCRefund {
    public void refund(double amount) {
        // create refund order
        // perform credit card refund
        // update refund order
    }
}
```

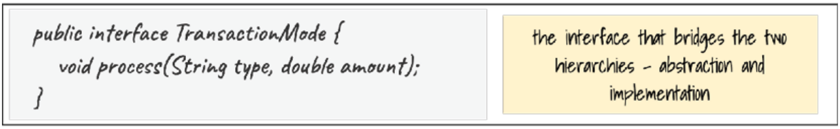
existing class implementations for processing class Credit Card payments and refunds

***Figure 11.6**: Existing implementations*

To avoid all these scenarios, the classes can be written in such a manner that they are only responsible for one task and delegate the rest to other classes.

Transaction mode

Figure 11.7 exhibits the **TransactionMode** interface that represents one of the hierarchies that we mentioned earlier. It exposes the functionality that handles integrations with payment service providers and processes monetary transfers.



```
public interface TransactionMode {
    void process(String type, double amount);
}
```

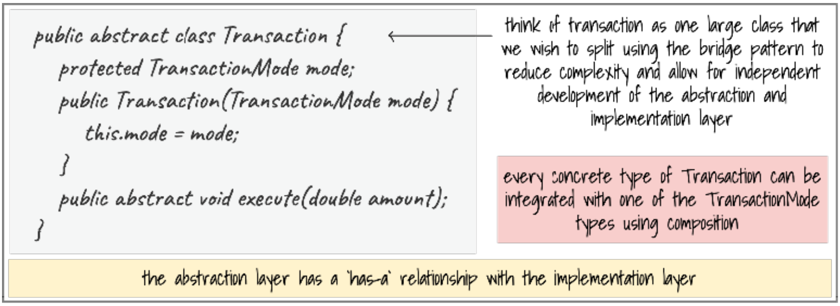
the interface that bridges the two hierarchies - abstraction and implementation

***Figure 11.7**: The TransactionMode interface*

It acts as a bridge between the two hierarchies and supports multiple behaviors by combining the two hierarchies.

Transaction

Figure 11.8 exhibits the abstract class **Transaction**. It represents the other hierarchy that takes part in the bridge design pattern. Clients interact with a concrete implementation of **Transaction** that delegates the responsibility of processing the money transfer to an implementation of **TransactionMode**.



```
public abstract class Transaction {
    protected TransactionMode mode;
    public Transaction(TransactionMode mode) {
        this.mode = mode;
    }
    public abstract void execute(double amount);
}
```

think of transaction as one large class that we wish to split using the bridge pattern to reduce complexity and allow for independent development of the abstraction and implementation layer

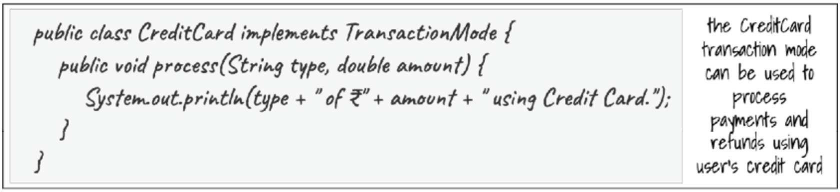
every concrete type of Transaction can be integrated with one of the TransactionMode types using composition

the abstraction layer has a 'has-a' relationship with the implementation layer

Figure 11.8: *Transaction*

Concrete transaction modes

The concrete implementations of the **TransactionMode** interface are responsible for processing payments and refunds by integrating with the payment service providers. Every transaction mode has a dedicated implementation. *Figure 11.9* exhibits the concrete implementation that processes payments and refunds via credit cards:

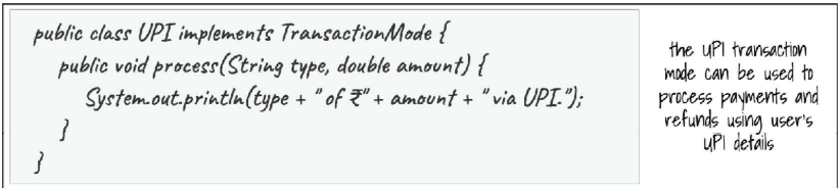


```
public class CreditCard implements TransactionMode {
    public void process(String type, double amount) {
        System.out.println(type + " of ₹" + amount + " using Credit Card.");
    }
}
```

the CreditCard transaction mode can be used to process payments and refunds using user's credit card

Figure 11.9: *Transaction mode is credit card*

Figure 11.10 exhibits another implementation of the **TransactionMode** interface that processes payments and refunds via the unified payment interface:



```
public class UPI implements TransactionMode {
    public void process(String type, double amount) {
        System.out.println(type + " of ₹" + amount + " via UPI.");
    }
}
```

the UPI transaction mode can be used to process payments and refunds using user's UPI details

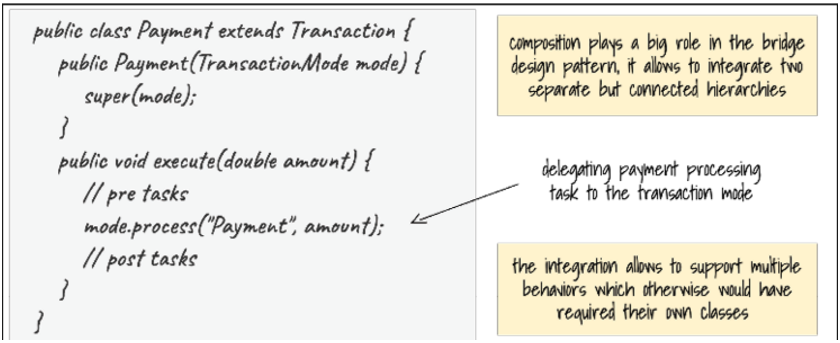
Figure 11.10: *Transaction mode: UPI*

Both these implementations accept type and amount information to process the money transfer.

Concrete transactions

Composition plays a big role in the bridge design pattern; it allows integrating two or more separate but connected hierarchies. The concrete implementations of the abstract class **Transaction** delegate requests to **TransactionMode** implementations.

Objects of one type can be injected into another using composition. *Figure 11.11* exhibits the use of composition to inject an object of type **TransactionMode** inside an object of type **Transaction** to process payments:



```
public class Payment extends Transaction {
    public Payment(TransactionMode mode) {
        super(mode);
    }
    public void execute(double amount) {
        // pre tasks
        mode.process("Payment", amount);
        // post tasks
    }
}
```

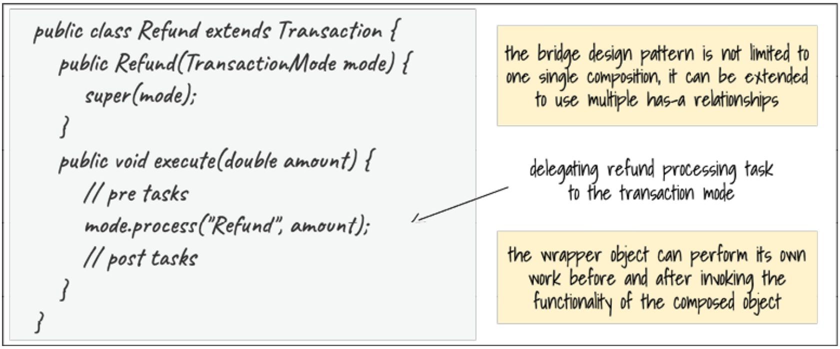
composition plays a big role in the bridge design pattern, it allows to integrate two separate but connected hierarchies

delegating payment processing task to the transaction mode

the integration allows to support multiple behaviors which otherwise would have required their own classes

***Figure 11.11**: Transaction: Payment*

Figure 11.12 exhibits another such implementation that processes refunds by delegating requests to the injected object:



```
public class Refund extends Transaction {
    public Refund(TransactionMode mode) {
        super(mode);
    }
    public void execute(double amount) {
        // pre tasks
        mode.process("Refund", amount);
        // post tasks
    }
}
```

the bridge design pattern is not limited to one single composition, it can be extended to use multiple has-a relationships

delegating refund processing task to the transaction mode

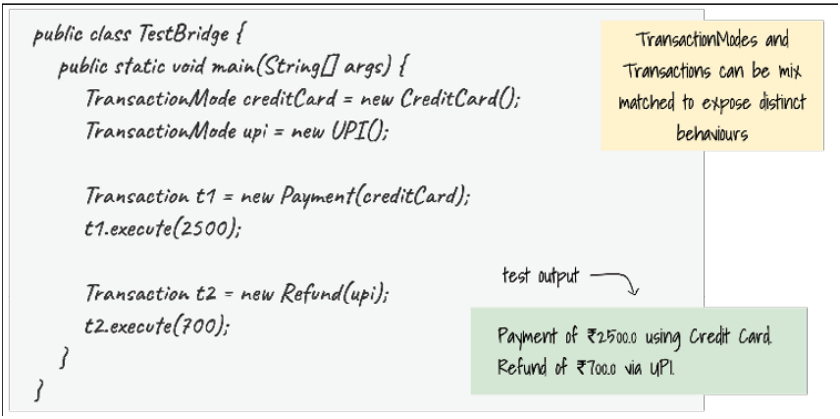
the wrapper object can perform its own work before and after invoking the functionality of the composed object

***Figure 11.12:** Transaction: Refund*

Both these implementations accept type and amount information to process the money transfer.

Testing the bridge

We have written a test class to verify the working of the bridge design pattern. Let us execute the same as exhibited in *Figure 11.13*:



```
public class TestBridge {
    public static void main(String[] args) {
        TransactionMode creditCard = new CreditCard();
        TransactionMode upi = new UPI();

        Transaction t1 = new Payment(creditCard);
        t1.execute(2500);

        Transaction t2 = new Refund(upi);
        t2.execute(700);
    }
}
```

TransactionModes and Transactions can be mix matched to expose distinct behaviours

test output

```
Payment of ₹2500.0 using Credit Card
Refund of ₹700.0 via UPI
```

***Figure 11.13**: Testing the bridge*

In the preceding example, we first create the transaction modes and then the types of transactions that use those transaction modes to process money transfers. The transaction modes are passed to the **Payment** and **Refund** transaction types during their construction. This approach keeps both these hierarchies separate and allows one to form distinct combinations. The test output reflects the responses received from the execution of those transactions.

Summary

The **bridge design pattern** is a structural design pattern that decouples an abstraction from its implementation, allowing both to vary independently. This is achieved by separating the abstraction (interface) from the implementation (concrete logic) and linking them via a bridge interface. It is beneficial when you want to avoid a permanent binding between an abstraction and its implementation, or when you need to support multiple variations of an object without creating a complex inheritance hierarchy.

In *Figure 11.14*, the bridge design pattern is reimaged with the help of the classes that we have used in our example:

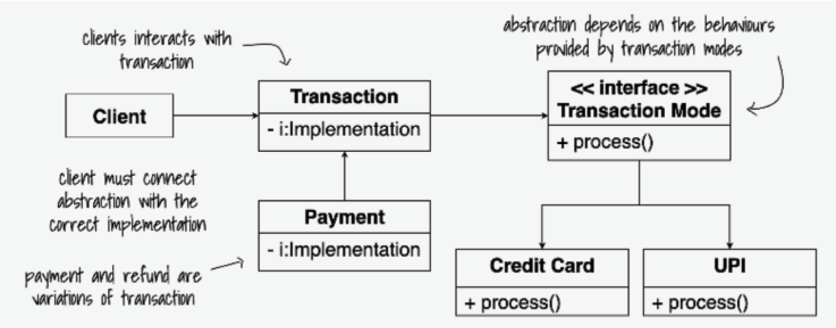

clients interacts with transaction
abstraction depends on the behaviours provided by transaction modes
Client
Transaction
- i:Implementation
<< interface >>
Transaction Mode
+ process()
client must connect abstraction with the correct implementation
Payment
- i:Implementation
payment and refund are variations of transaction
Credit Card
+ process()
UPI
+ process()

***Figure 11.14**: Bridge design pattern—defined*

Advantages and drawbacks

It is time to make a note of some of the benefits and drawbacks of the bridge design pattern. Let us start with the benefits first followed by the drawbacks:

- You can modify or extend either side (abstraction or implementation) independently without affecting the other.
- Since responsibilities are split, the code becomes easier to understand, modify, and test.
- You can add new abstractions and implementations without changing existing code.
- Avoids a combinatorial explosion of classes that happens when trying to subclass all possible combinations of features.

The drawbacks of the bridge design pattern are:

- Introducing additional layers of abstraction and interfaces can make the system more complex and harder to understand for newcomers.
- For small or tightly-coupled systems, using the bridge pattern may be unnecessary and add avoidable abstraction overhead.
- Since the actual implementation is separated and dynamically associated, it can be harder to trace execution

flow or debug.

Usage

Some of the scenarios relevant to the bridge design pattern are:

- **GUI frameworks**: A window (abstraction) delegates rendering to the respective OS-specific implementation, allowing the same app to run on different platforms.
- **Messaging system**: Messages can be sent via different channels and have different urgency levels.

Conclusion

The bridge design pattern is a powerful structural pattern that promotes flexibility, scalability, and clean separation of concerns. By decoupling abstractions from their implementations, it enables developers to independently vary both sides of the abstraction-implementation relationship. This results in code that is easier to maintain, extend, and test, especially in systems with evolving or diverse feature sets.

However, like all patterns, the bridge is best used when its benefits, such as reducing code duplication across similar classes, outweigh the complexity it introduces. When applied thoughtfully, it leads to a well-organized architecture that stands the test of change and growth.

In the next chapter, we will learn about the filter design principles and how they can be utilized to filter a collection of objects based on certain criteria.

Questions

In this section, we will sharpen our brains by indulging in some fun brain activities. Let us get started.

Find the odd one out

1. Purpose of the bridge design pattern:
    a. Decouples abstraction from implementation

    b. Encourages tight coupling between components
    c. Enables independent evolution of abstraction and implementation
    d. Helps avoid class explosion when combining multiple features
2. Which of these is not typically a participant in the bridge pattern?
    a. Abstraction
    b. Refined abstraction
    c. Adapter
    d. Implementor
3. Which of the following is not an advantage of using the bridge pattern?
    a. Promotes code reusability
    b. Simplifies complex APIs into single inheritance chains
    c. Supports open/closed principle
    d. Reduces code duplication
4. Which of the following use cases is least likely to benefit from the bridge pattern?
    a. Drawing application with different shapes and rendering APIs
    b. A notification system with multiple channels (Email, SMS, Push)
    c. A mathematical library with fixed formulas and static functions
    d. A payment system with multiple gateways and transaction types

Select two correct answers

1. Which of the following are core benefits of the bridge design pattern?
    a. Increases tight coupling between abstraction and implementation

   - b. Allows abstraction and implementation to vary independently
   - c. Promotes inheritance over composition
   - d. Reduces code duplication in large class hierarchies
2. Which scenarios are good candidates for applying the bridge pattern?
   - a. A system where UI themes need to work across different platforms
   - b. A program needing to convert JSON to XML format
   - c. A reporting tool supporting multiple report types and export formats
   - d. A fixed mathematical calculator with hardcoded functions

Answers

This section provides the answers to the given questions.

Find the odd one out

1. Encourages tight coupling between components
2. Adapter
3. Simplifies complex APIs into single inheritance chains
4. A mathematical library with fixed formulas and static functions

Select two correct answers

1. Allows abstraction and implementation to vary independently, reduce code duplication in large class hierarchies
2. A system where UI themes need to work across different platforms promotes inheritance over composition

Join our Discord space

Join our Discord workspace for latest updates, offers, tech happenings around the world, new releases, and sessions with the authors:

https://discord.bpbonline.com

CHAPTER 12

Applying Filters

Introduction

Data analysis is at the core of many business operations, and the insights gathered by performing such an activity help to steer the business forward. For example, the analysis done to understand the viewership pattern on a streaming platform across multiple categories, based on ratings and watch time of available videos. Refer to the following figure:

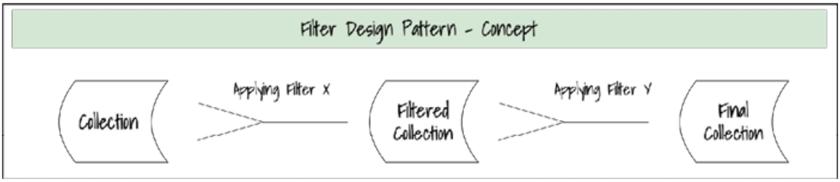

Filter Design Pattern - Concept
Collection
Applying Filter X
Filtered Collection
Applying Filter Y
Final Collection

***Figure 12.1**: Filters*

Performing analytical operations on such large datasets requires segregation and filtering of data at multiple steps, and usually involves many attributes and variations of data. The filter design pattern provides a simple and efficient way to filter the dataset and reduce it to gain useful information.

Structure

This chapter covers the following topics:

- Filter design pattern
- Implementing filter pattern
- Summary
- Advantages and drawbacks
- Usage

Objectives

The objective is to acquire knowledge about the filter design pattern and its use in reducing and reshaping a large dataset while extracting useful insights from it. The filter design pattern provides a mechanism that allows narrowing down the contents of a large dataset based on the filters that are applied to it to produce a smaller dataset with each application of a filter.

After finishing this chapter, the readers will be able to understand the working of the filter design pattern and how it can be utilized to filter out the noise from a dataset to gather the required information.

Filter design pattern

The filter design pattern is a structural design pattern that can be used to filter a collection of objects based on some criteria. It is also known as the **criteria design pattern**, as shown in the following figure:

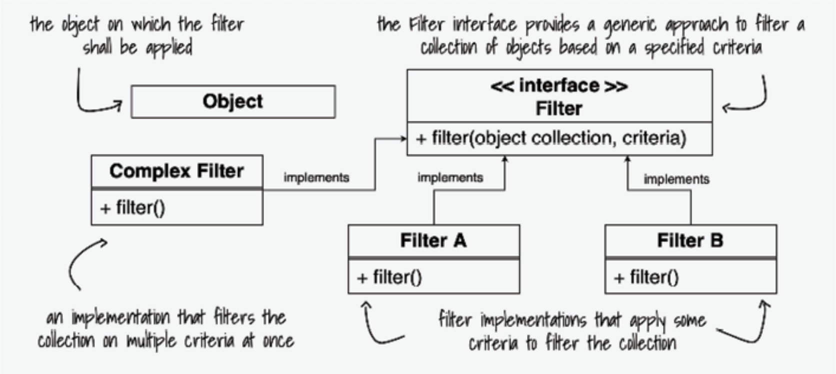

the object on which the filter shall be applied
Object
the Filter interface provides a generic approach to filter a collection of objects based on a specified criteria
<< interface >>
Filter
+ filter(object collection, criteria)
Complex Filter
+ filter()
implements
implements
implements
Filter A
+ filter()
Filter B
+ filter()
an implementation that filters the collection on multiple criteria at once
filter implementations that apply some criteria to filter the collection

Figure 12.2: Filter design pattern

This pattern advocates for modular development and single responsibility. It is possible to write distinct filtering logics in their own classes and use them interchangeably. Each implementation of a filter is reusable and can be applied to different datasets. Since they do not have a state, they can be used without the risk of data loss or manipulation due to concurrent or parallel usage.

Filter implementations are available in both specific and generic forms. A specific implementation involves filtering a collection of objects based on a specific field value, like sorting pizza orders based on their status. The only input required is the value of the field. On the flip side, a generic filter accepts both the object field and the filter value as input.

Filter implementations that mimic logical operations such as *and, not,* and *or* can be advantageous when two or more criteria determine the outcome of the filter operation. *Figure 12.3* illustrates two scenarios where logical operations come in handy. In the first example, a filter is used to locate orders that were successful and placed on New Year's evening. In the second example, an OR filter is used to find orders that the customer has cancelled or returned.

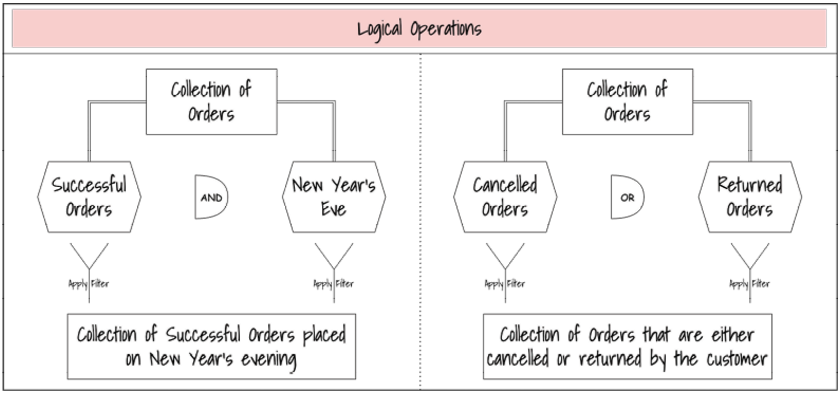

Logical Operations
Collection of Orders
Successful Orders
AND
New Year's Eve
Apply Filter
Apply Filter
Collection of Successful Orders placed on New Year's evening
Collection of Orders
Cancelled Orders
OR
Returned Orders
Apply Filter
Apply Filter
Collection of Orders that are either cancelled or returned by the customer

Figure 12.3: Logical operations

Implementing the filter pattern

This chapter will provide an effective explanation of the filter design pattern through the example we will walk you through. We will use the filter design pattern for applying filters on large sets of orders placed on a website to gain valuable insights, such as filtering the orders based on minimum order value and their status.

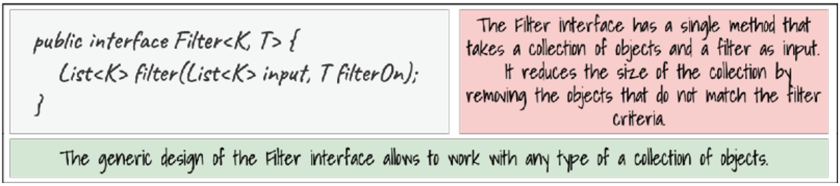
We will also check out filter implementations that behave like a logical operation, for example, applying only one of the passed filters just like a logical OR, or applying two operations together like a logical AND.

The filter design pattern is like a where clause in a **Structured Query Language** (**SQL**) query, but for a list of objects. By implementing specific criteria, you can use a where clause to narrow down the number of query results. Similarly, a filter implementation reduces the size of a collection of objects by applying some criteria. Filters can also be used in a sequence to narrow down the result set after each application, allowing for more granular insights on the same set of data.

We will learn about the implementation of the filter design pattern and how it can be used to gain valuable insights from large sets of data.

Filter interface

Abstraction and polymorphism are at the core of most of the design patterns, and the same goes for the filter pattern. *Figure 12.4* illustrates the interface that allows a class to function as a filter when implemented:



```
public interface Filter<K, T> {
    List<K> filter(List<K> input, T filterOn);
}
```

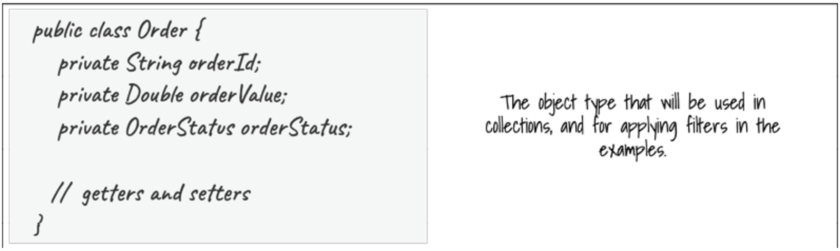
The Filter interface has a single method that takes a collection of objects and a filter as input. It reduces the size of the collection by removing the objects that do not match the filter criteria.

The generic design of the Filter interface allows to work with any type of a collection of objects.

Figure 12.4*: Filter interface*

The interface declaration is a generic one and allows for a wide variety of implementations. The filter method takes two inputs: a collection of objects and the criteria that should be used to filter the collection. Both these inputs use generics and, therefore, can be replaced with any concrete types. This approach makes it simple and efficient for the developers to write any number of filters and use them interchangeably when required.

Figure 12.5 exhibits the object type that will be used for building a collection of objects for the purpose of applying filters to learn and understand the filter design pattern. The examples will make use of only a few data members of the object type to ensure clarity and purpose.



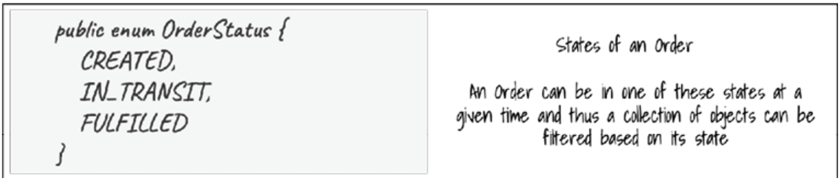
```
public class Order {
    private String orderId;
    private Double orderValue;
    private OrderStatus orderStatus;

   // getters and setters
}
```

The object type that will be used in collections, and for applying filters in the examples.

***Figure 12.5**: Collection object*

You can think of an order as an object that stores information related to an actual order placed on an **e-commerce** website. A variety of useful information can be fetched from a collection of objects based on certain criteria, for example, get all the failed orders within a timeframe, or get all the pending orders that were paid via credit cards.

One of the data members of the **Order** class uses another type, an enum, to limit the values that it can contain. *Figure 12.6* exhibits the possible states that an order can be in. These values are very useful when filters are applied to the collection of objects:



```
public enum OrderStatus {
    CREATED,
    IN_TRANSIT,
    FULFILLED
}
```

States of an Order

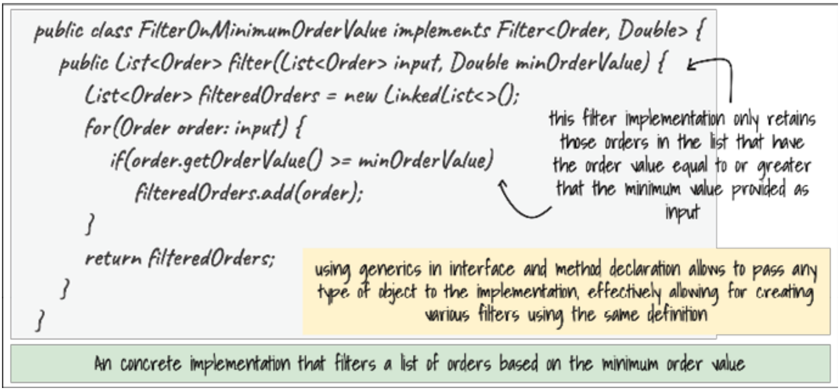
An Order can be in one of these states at a given time and thus a collection of objects can be filtered based on its state

***Figure 12.6**: Order status*

Simple filters

The concrete implementations of the **Filter** interface contain the business logic that is applied to the collection of objects to filter and reduce it. Post the application, only those objects that meet the criteria defined in the applied filter implementation remain in the collection, and the rest of the objects are discarded.

Figure 12.7 exhibits one such filter implementation that takes a collection as an input and filters it to find all the orders that have a minimum order value, not less than what is passed in the second input.



```
public class FilterOnMinimumOrderValue implements Filter<Order, Double> {
    public List<Order> filter(List<Order> input, Double minOrderValue) {
        List<Order> filteredOrders = new LinkedList<>();
        for(Order order: input) {
            if(order.getOrderValue() >= minOrderValue)
                filteredOrders.add(order);
        }
        return filteredOrders;
    }
}
```

this filter implementation only retains those orders in the list that have the order value equal to or greater that the minimum value provided as input

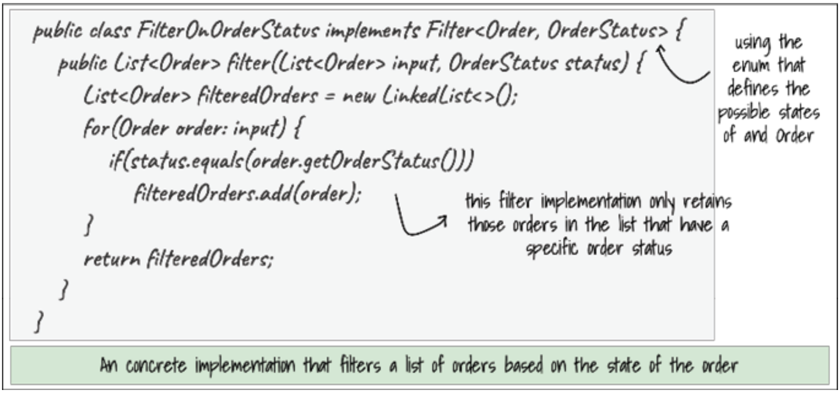
using generics in interface and method declaration allows to pass any type of object to the implementation, effectively allowing for creating various filters using the same definition

An concrete implementation that filters a list of orders based on the minimum order value

Figure 12.7: *Minimum order value filter*

This is a simple yet powerful example of the filter design pattern. The filter design pattern allows for segregation of filtration logic into separate classes, making it easier to maintain and update without affecting other filter implementations.

Another such implementation that allows filtering a collection of orders based on their status is exhibited in *Figure 12.8*:



```
public class FilterOnOrderStatus implements Filter<Order, OrderStatus> {
    public List<Order> filter(List<Order> input, OrderStatus status) {
        List<Order> filteredOrders = new LinkedList<>();
        for(Order order: input) {
            if(status.equals(order.getOrderStatus()))
                filteredOrders.add(order);
        }
        return filteredOrders;
    }
}
```

using the enum that defines the possible states of and Order

this filter implementation only retains those orders in the list that have a specific order status

An concrete implementation that filters a list of orders based on the state of the order

Figure 12.8: *Order status filter*

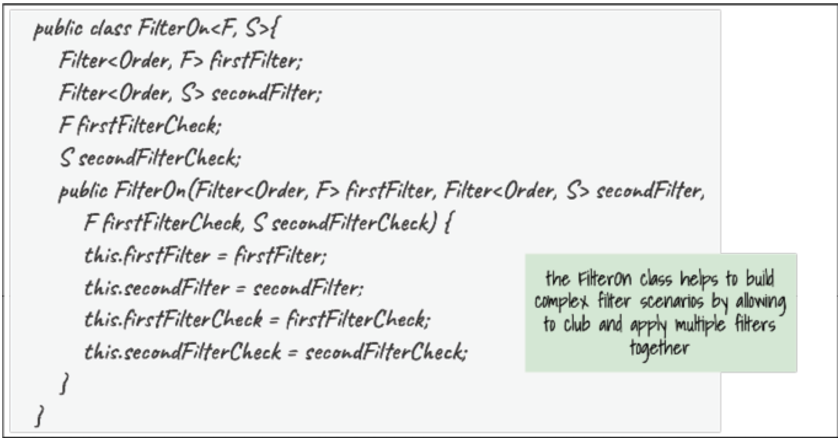
In this implementation, each order is checked against a certain state, and only those orders are retained that are in that state, while all others are discarded.

The response type of the **filter** method is the same as the type of the object that the collection contains. This allows for the sequential application of filters to the collection. In other words, filters can be applied on a collection one after another to fetch the desired objects from the collection.

Complex filters

Filter implementations can also be made to filter collections based on multiple criteria instead of just one, as we have seen so far. While multiple filters can be applied in sequence to achieve the desired output, there are certain scenarios where the filtration logic must rely on multiple fields, e.g., search for all the orders in a collection whose status is *success* or *pending*.

Simple filters can also be made to run as one single unit using complex filters. In other words, filters can be grouped together as exhibited in *Figure 12.9*. A class like the one shown in the following figure can encapsulate multiple filter implementations together. This ensures that the filters that must run together are not accidentally left out.



```
public class FilterOn<F, S>{
    Filter<Order, F> firstFilter;
    Filter<Order, S> secondFilter;
    F firstFilterCheck;
    S secondFilterCheck;
    public FilterOn(Filter<Order, F> firstFilter, Filter<Order, S> secondFilter,
        F firstFilterCheck, S secondFilterCheck) {
        this.firstFilter = firstFilter;
        this.secondFilter = secondFilter;
        this.firstFilterCheck = firstFilterCheck;
        this.secondFilterCheck = secondFilterCheck;
    }
}
```

the FilterOn class helps to build complex filter scenarios by allowing to club and apply multiple filters together

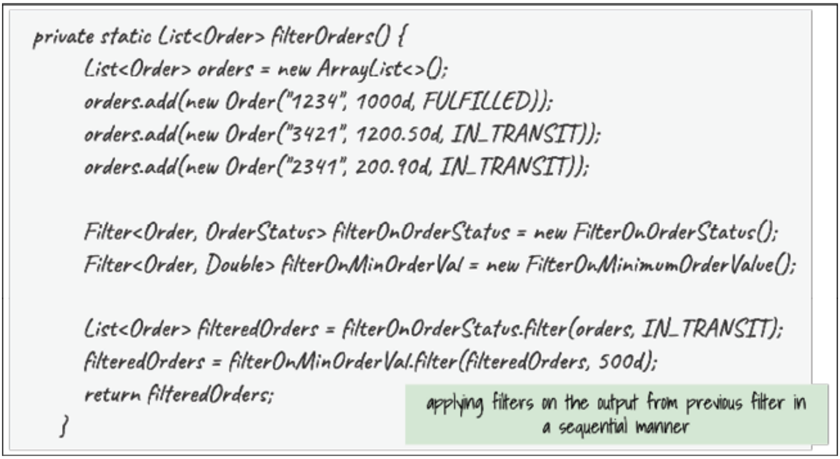
Figure 12.9*: Encapsulating filter implementations*

Using complex implementations, filters can be combined using logical operations, and there is no limit on the number of filters that can be combined to perform a single task. In *Figure 12.9*, the wrapper class takes only two filters as input, but it can be changed to take a list of filters instead. With that change, any number of filters can be passed to the wrapper and used to perform more complex logical operations.

Logical operations

As discussed in the preceding section, filter implementations can make use of logical operations to apply multiple criteria on a collection, and we will learn more about this in this section.

When the filters are clubbed together and called in a sequence, as exhibited in *Figure 12.10*, the collection is further reduced due to the reapplication of filters on already filtered data:



```
private static List<Order> filterOrders() {
        List<Order> orders = new ArrayList<>();
        orders.add(new Order("1234", 1000d, FULFILLED));
        orders.add(new Order("3421", 1200.50d, IN_TRANSIT));
        orders.add(new Order("2341", 200.90d, IN_TRANSIT));

        Filter<Order, OrderStatus> filterOnOrderStatus = new FilterOnOrderStatus();
        Filter<Order, Double> filterOnMinOrderVal = new FilterOnMinimumOrderValue();

        List<Order> filteredOrders = filterOnOrderStatus.filter(orders, IN_TRANSIT);
        filteredOrders = filterOnMinOrderVal.filter(filteredOrders, 500d);
        return filteredOrders;
    }
```

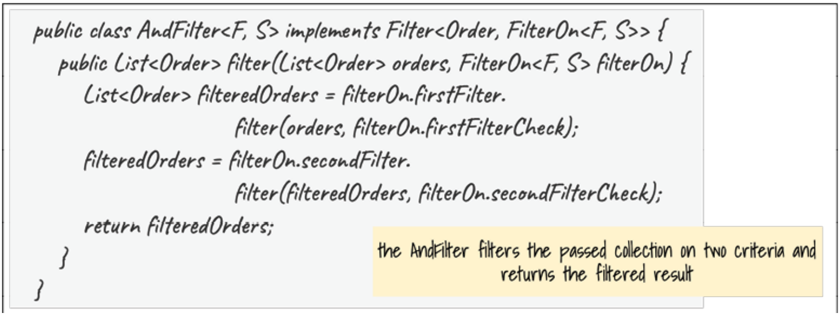
applying filters on the output from previous filter in a sequential manner

***Figure 12.10**: Filters called in a sequence*

This approach, by applying the filters one after the other, where the output from the previous filter is used as input for the next one, is like applying an **AND** operation between the two filters and reducing the collection based on both criteria. Synchronous application of filters always results in an implicit **AND** operation.

Logical operation: AND

Another way to imitate the **AND** operation is by implementing a filter that can perform filtration based on more than one criterion, much like encapsulating the sequential application inside the implementation, as exhibited in *Figure 12.11*. This is a generic implementation that accepts two filters and applies them sequentially on the collection passed as input.



```
public class AndFilter<F, S> implements Filter<Order, FilterOn<F, S>> {
    public List<Order> filter(List<Order> orders, FilterOn<F, S> filterOn) {
        List<Order> filteredOrders = filterOn.firstFilter.
                        filter(orders, filterOn.firstFilterCheck);
        filteredOrders = filterOn.secondFilter.
                        filter(filteredOrders, filterOn.secondFilterCheck);
        return filteredOrders;
    }
}
```

the AndFilter filters the passed collection on two criteria and returns the filtered result

Figure 12.11: *Logical operation: AND*

Both the approaches mentioned earlier use the same execution pattern, and because of that, an explicit filter implementation that performs the **AND** operation does not seem very useful, but there are certain advantages to using a dedicated filter implementation:

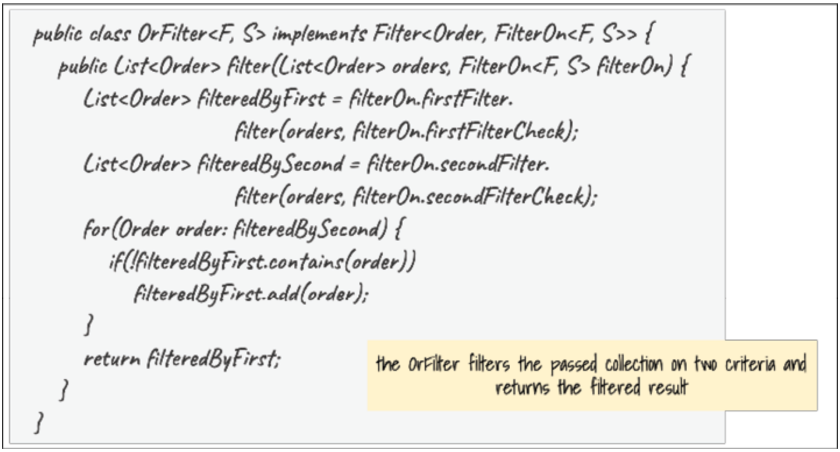
- It encapsulates the execution logic, allowing for reuse.
- It ensures that the filters are always applied whenever the implementation is invoked.
- It also helps with simplifying the filtration logic by making it more readable and expressive.

Logical operation: OR

The **OR** operation is another logical operation that can be performed while filtering a collection of objects. While the implicit behavior while applying filters sequentially mimics the **AND** operation, there is no direct way to implement logical **OR**. Therefore, the filters that can execute such a behavior become even more useful.

A logical **OR** operation allows filtering the collection when either of the criteria is met, unlike the **AND** operation, where both criteria must be met. For example, a collection of orders can be filtered based on whether they are successful or were delivered before yesterday. Another such example could be to find all the orders that are either paid using credit cards or the order amount exceeds a certain threshold.

Figure 12.12 exhibits a generic implementation that accepts two filters and applies them both to the collection as per the filter execution logic:



```
public class OrFilter<F, S> implements Filter<Order, FilterOn<F, S>> {
    public List<Order> filter(List<Order> orders, FilterOn<F, S> filterOn) {
        List<Order> filteredByFirst = filterOn.firstFilter.
                            filter(orders, filterOn.firstFilterCheck);
        List<Order> filteredBySecond = filterOn.secondFilter.
                            filter(orders, filterOn.secondFilterCheck);
        for(Order order: filteredBySecond) {
            if(!filteredByFirst.contains(order))
                filteredByFirst.add(order);
        }
        return filteredByFirst;
    }
}
```

the OrFilter filters the passed collection on two criteria and returns the filtered result

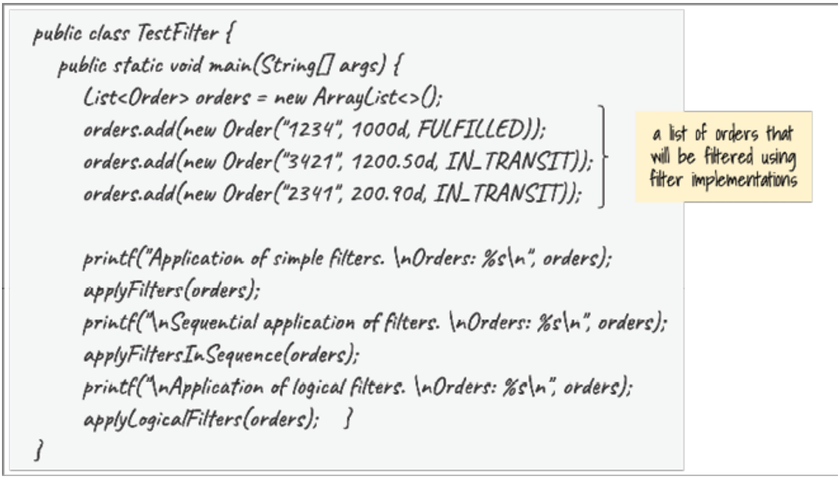
***Figure 12.12**: Logical operation: OR*

More filters can be clubbed together to perform more complex logical operations by using the wrapper class exhibited in *Figure 12.9*. Since the wrapper class expects objects of type filter as parameters, the filters that execute logical operations can also be passed to it, allowing for multiple combinations of logical operations. For example, we can filter a collection of objects based on the following criteria. The status of orders should be *dispatched,* or *success*, and the *order amount* should be greater than *1000*.

Using the preceding approach, a collection can be filtered by combining multiple such criteria, simple or complex ones, and applying those to the collection.

Testing the filters

We have written a test class to verify the working of the filter design pattern. Let us execute the same as shown in *Figure 12.13*:

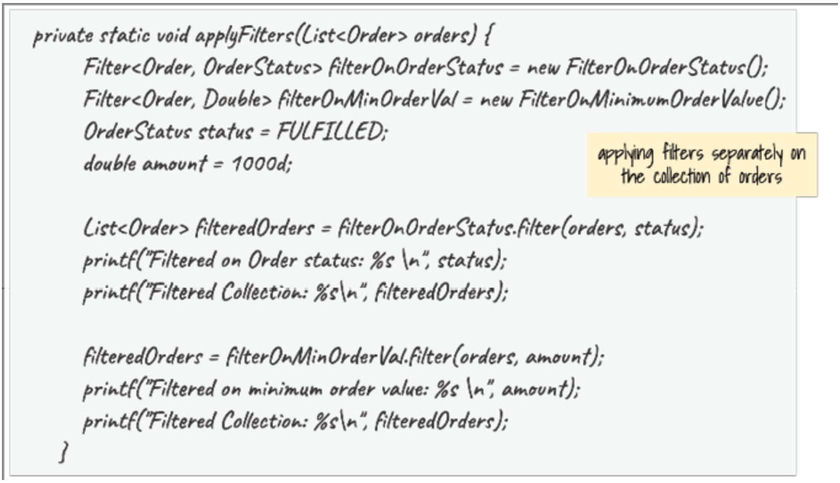


```
public class TestFilter {
   public static void main(String[] args) {
      List<Order> orders = new ArrayList<>();
      orders.add(new Order("1234", 1000d, FULFILLED));
      orders.add(new Order("3421", 1200.50d, IN_TRANSIT));
      orders.add(new Order("2341", 200.90d, IN_TRANSIT));

      printf("Application of simple filters. \nOrders: %s\n", orders);
      applyFilters(orders);
      printf("\nSequential application of filters. \nOrders: %s\n", orders);
      applyFiltersInSequence(orders);
      printf("\nApplication of logical filters. \nOrders: %s\n", orders);
      applyLogicalFilters(orders);    }
}
```

a list of orders that will be filtered using filter implementations

Figure 12.13: Testing the filters

In the preceding example, a collection is being initialized and populated with order objects. Next, this collection is passed to three different methods to test the filter implementations. *Figure 12.14* exhibits the first method definition that applies simple filters to the collection of order objects:

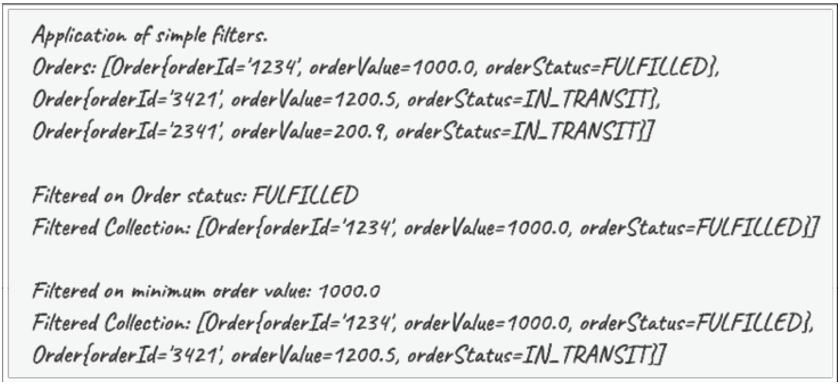


```
private static void applyFilters(List<Order> orders) {
      Filter<Order, OrderStatus> filterOnOrderStatus = new FilterOnOrderStatus();
      Filter<Order, Double> filterOnMinOrderVal = new FilterOnMinimumOrderValue();
      OrderStatus status = FULFILLED;
      double amount = 1000d;

      List<Order> filteredOrders = filterOnOrderStatus.filter(orders, status);
      printf("Filtered on Order status: %s \n", status);
      printf("Filtered Collection: %s\n", filteredOrders);

      filteredOrders = filterOnMinOrderVal.filter(orders, amount);
      printf("Filtered on minimum order value: %s \n", amount);
      printf("Filtered Collection: %s\n", filteredOrders);
   }
```

applying filters separately on the collection of orders

Figure 12.14: Applying simple filters

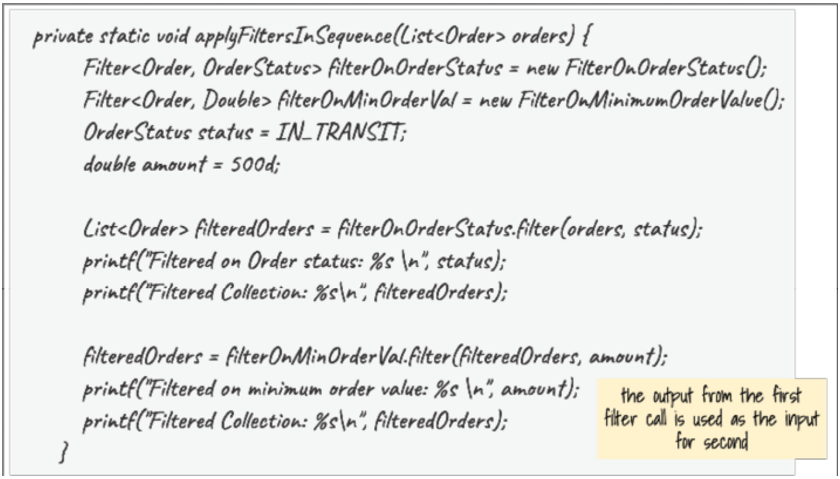
Figure 12.15 exhibits the output of applying the preceding filters on the collection of objects. The first filter applies the criteria on order status and finds all such orders that are fulfilled. The second filter applies the criteria to the minimum order amount.



```
Application of simple filters.
Orders: [Order{orderId='1234', orderValue=1000.0, orderStatus=FULFILLED},
Order{orderId='3421', orderValue=1200.5, orderStatus=IN_TRANSIT},
Order{orderId='2341', orderValue=200.9, orderStatus=IN_TRANSIT}]

Filtered on Order status: FULFILLED
Filtered Collection: [Order{orderId='1234', orderValue=1000.0, orderStatus=FULFILLED}]

Filtered on minimum order value: 1000.0
Filtered Collection: [Order{orderId='1234', orderValue=1000.0, orderStatus=FULFILLED},
Order{orderId='3421', orderValue=1200.5, orderStatus=IN_TRANSIT}]
```

***Figure 12.15**: Result of applying simple filters*

The next method tests the sequential application of filters on the object collection. It applies the same set of filters but with different criteria. *Figure 12.16* exhibits the method definition where the filters are applied one after another on the same collection. The output from the previous filter is used as the input for the next filter implementation.

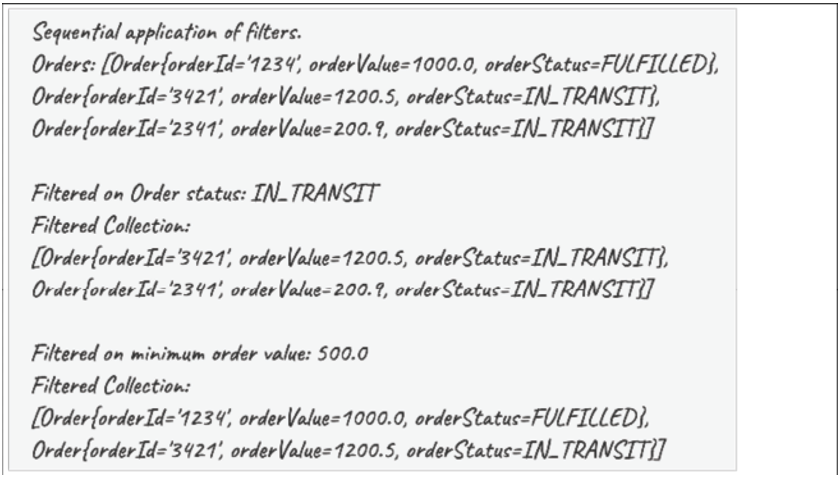


```
private static void applyFiltersInSequence(List<Order> orders) {
        Filter<Order, OrderStatus> filterOnOrderStatus = new FilterOnOrderStatus();
        Filter<Order, Double> filterOnMinOrderVal = new FilterOnMinimumOrderValue();
        OrderStatus status = IN_TRANSIT;
        double amount = 500d;

        List<Order> filteredOrders = filterOnOrderStatus.filter(orders, status);
        printf("Filtered on Order status: %s \n", status);
        printf("Filtered Collection: %s\n", filteredOrders);

        filteredOrders = filterOnMinOrderVal.filter(filteredOrders, amount);
        printf("Filtered on minimum order value: %s \n", amount);
        printf("Filtered Collection: %s\n", filteredOrders);
    }
```

the output from the first filter call is used as the input for second

***Figure 12.16**: Sequential application of filters*

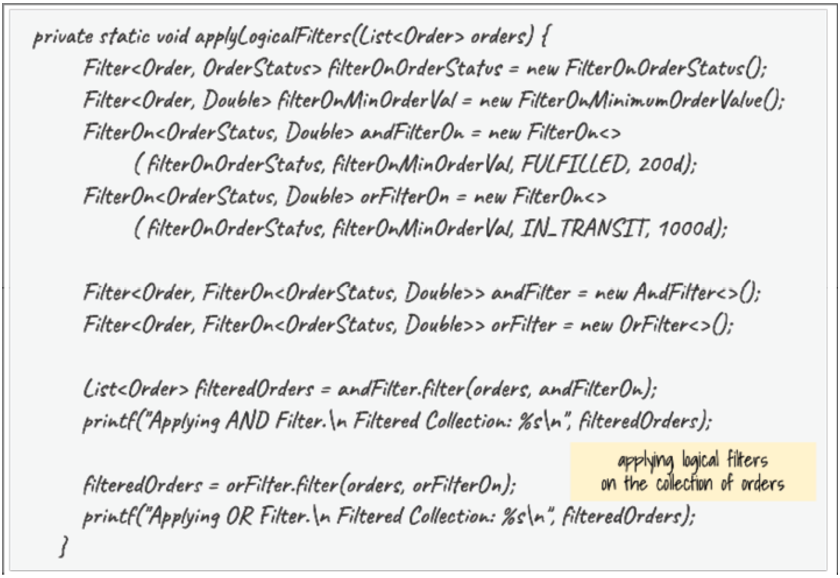
Figure 12.17 exhibits the result of applying filters in a sequence. The first filter finds all such orders that are in transit but not yet delivered:



```
Sequential application of filters.
Orders: [Order{orderId='1234', orderValue=1000.0, orderStatus=FULFILLED},
Order{orderId='3421', orderValue=1200.5, orderStatus=IN_TRANSIT},
Order{orderId='2341', orderValue=200.9, orderStatus=IN_TRANSIT}]

Filtered on Order status: IN_TRANSIT
Filtered Collection:
[Order{orderId='3421', orderValue=1200.5, orderStatus=IN_TRANSIT},
Order{orderId='2341', orderValue=200.9, orderStatus=IN_TRANSIT}]

Filtered on minimum order value: 500.0
Filtered Collection:
[Order{orderId='1234', orderValue=1000.0, orderStatus=FULFILLED},
Order{orderId='3421', orderValue=1200.5, orderStatus=IN_TRANSIT}]
```

Figure 12.17: *Result of sequential application of filters*

The last method, as exhibited in *Figure 12.18*, tests the use of logical operations while applying filters on a collection of orders. It applies both the **OR** and **AND** operations on a collection of orders.

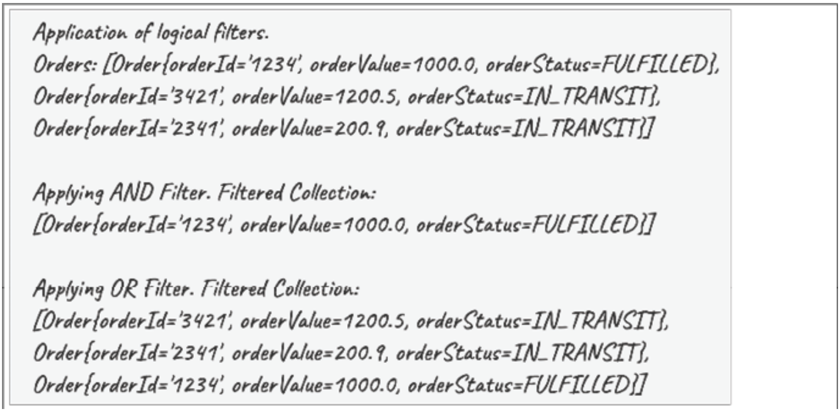


```
private static void applyLogicalFilters(List<Order> orders) {
        Filter<Order, OrderStatus> filterOnOrderStatus = new FilterOnOrderStatus();
        Filter<Order, Double> filterOnMinOrderVal = new FilterOnMinimumOrderValue();
        FilterOn<OrderStatus, Double> andFilterOn = new FilterOn<>
                ( filterOnOrderStatus, filterOnMinOrderVal, FULFILLED, 200d);
        FilterOn<OrderStatus, Double> orFilterOn = new FilterOn<>
                ( filterOnOrderStatus, filterOnMinOrderVal, IN_TRANSIT, 1000d);

        Filter<Order, FilterOn<OrderStatus, Double>> andFilter = new AndFilter<>();
        Filter<Order, FilterOn<OrderStatus, Double>> orFilter = new OrFilter<>();

        List<Order> filteredOrders = andFilter.filter(orders, andFilterOn);
        printf("Applying AND Filter.\n Filtered Collection: %s\n", filteredOrders);

        filteredOrders = orFilter.filter(orders, orFilterOn);
        printf("Applying OR Filter.\n Filtered Collection: %s\n", filteredOrders);
    }
```

applying logical filters on the collection of orders

Figure 12.18: *Filters—logical operations*

Figure 12.19 exhibits the result of applying logical filters on the collection. The first filter applies the **AND** operation, and the second one applies the **OR** operation:



```
Application of logical filters.
Orders: [Order{orderId='1234', orderValue=1000.0, orderStatus=FULFILLED},
Order{orderId='3421', orderValue=1200.5, orderStatus=IN_TRANSIT},
Order{orderId='2341', orderValue=200.9, orderStatus=IN_TRANSIT}]

Applying AND Filter. Filtered Collection:
[Order{orderId='1234', orderValue=1000.0, orderStatus=FULFILLED}]

Applying OR Filter. Filtered Collection:
[Order{orderId='3421', orderValue=1200.5, orderStatus=IN_TRANSIT},
Order{orderId='2341', orderValue=200.9, orderStatus=IN_TRANSIT},
Order{orderId='1234', orderValue=1000.0, orderStatus=FULFILLED}]
```

Figure 12.19*: Result—logical operations*

Summary

The filter design pattern is a structural design pattern that allows filtering a collection of objects using different criteria and chaining them in a decoupled way. It allows filtering a collection of objects using multiple criteria without hardcoding the logic in a single class.

In *Figure 12.20*, the filter design pattern is reimagined with the help of classes that we have used in our example:

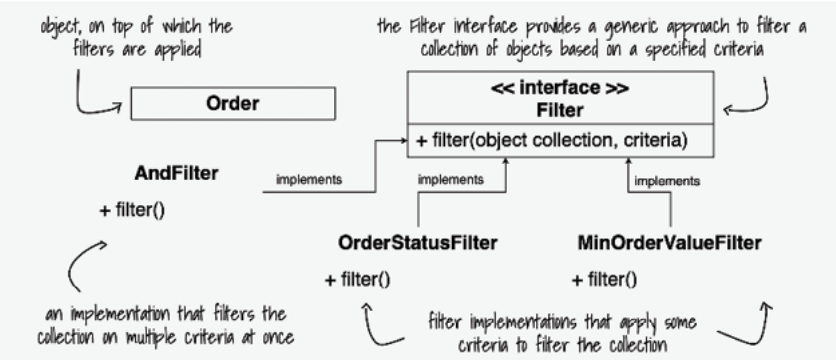

object, on top of which the filters are applied
Order
the Filter interface provides a generic approach to filter a collection of objects based on a specified criteria
<< interface >>
Filter
+ filter(object collection, criteria)
AndFilter
+ filter()
implements
implements
implements
OrderStatusFilter
+ filter()
MinOrderValueFilter
+ filter()
an implementation that filters the collection on multiple criteria at once
filter implementations that apply some criteria to filter the collection

Figure 12.20*: Filter design pattern—defined*

It utilizes abstraction and promotes reusability by encapsulating the filtration logic in separate classes. It allows combining filters using logical operations, like **OR** and **AND**. It also promotes the open/closed principle and ensures that the filters are open to extension but closed to modification.

Advantages and drawbacks

It is time to make a note of some of the benefits and drawbacks of the filter design pattern. Let us discuss the benefits first followed by the drawbacks:

- The filter pattern is highly extensible by design.
- Each filtering criterion is encapsulated in its own class.
- Filters can be combined using logical operations.
- It promotes reusability.

The drawbacks of the pipeline design pattern are:

- With multiple implementations, it can lead to many small classes.
- Basic loops can sometimes be a better choice in comparison to the filter pattern, especially when the criteria checks are very simple.

Usage

The following list highlights some of the scenarios that are relevant to the filter design pattern:

- For filtering data by applying different search filters, for example, order search filters.
- **User interface** (**UI**) dashboards and rule engines are one of the important usage examples for the filter design pattern.

Conclusion

The filter design pattern allows filtering a collection of objects and returns a sub-collection that matches all the criteria that are applied to that collection. It supports the open/close principle and is highly extensible. It promotes extensibility and reusability by encapsulating the filtration logic inside separate classes. The filters can be easily applied to reduce the original collection to a smaller subset that has only filtered objects.

In the next chapter, we will learn about the composite design pattern and how it can be used to treat individual objects and a group of objects uniformly.

Questions

In this section, we will sharpen our brains by indulging in some fun brain activities. Let us get started.

Find the odd one out

1. Filter design pattern:
    a. Promotes separation of concern
    b. Promotes open/close principle
    c. Is a behavioral design pattern
    d. Allows for combining filters
2. Filter implementations:
    a. Can be reused
    b. Can filter Map objects
    c. Can only be used for OR logical operation
    d. Can be applied on any collection type

Select two correct answers

1. A filter:
    a. Is extensible
    b. Can be combined to perform logical operations
    c. Is open for modification
    d. Can only work with lists and not sets
2. Filters that implement logical operations:
    a. Can only be used for two types (OR, AND)
    b. Are extensible

c. Are mutable
d. Increase complexity

Answers

This section provides the answers to the questions given.

Find the odd one out

1. Is a behavioral design pattern
2. Can only be used for OR logical operation

Select two correct answers

1. Is open for modification, can only work with lists and not sets
2. Are mutable, increase complexity

CHAPTER 13

Composite

Introduction

Sometimes in software, we deal with structures where individual items and groups of items need to be treated the same way—like files inside folders, or shapes inside a drawing. The **composite design pattern** helps us do exactly that. It lets us build objects in a tree-like hierarchy where both single objects and collections share a common interface. *Figure 13.1* exhibits this concept in the form of a diagram. The complex structure can contain other simple or complex structures. This allows for extending the hierarchy tree to an indefinite length.

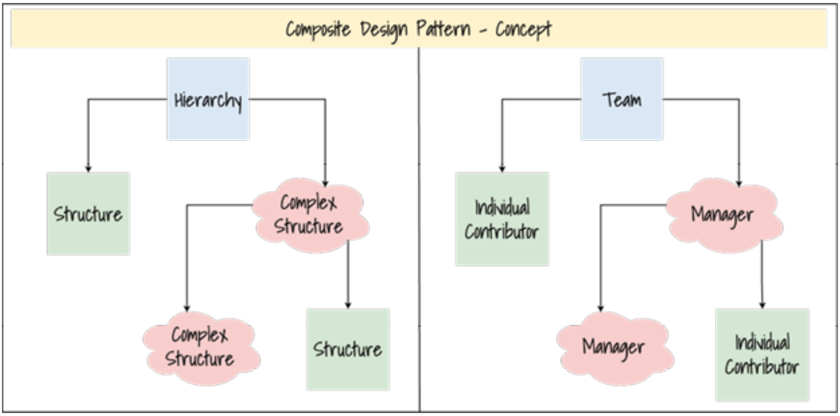

Composite Design Pattern - Concept
Hierarchy
Structure
Complex Structure
Complex Structure
Structure
Team
Individual Contributor
Manager
Manager
Individual Contributor

***Figure 13.1**: Composite design pattern—concept*

This means we can work with them uniformly without worrying whether we are dealing with one item or a whole group. The result is cleaner, more flexible code that mirrors how we naturally organize things in the real world.

Structure

This chapter covers the following topics:

- Composite design pattern
- Implementing the composite pattern
- Summary
- Advantages and drawbacks
- Usage

Objectives

By the end of this chapter, the readers will learn about composite design pattern that simplifies working with an object and a collection of those objects. It will allow them to work with similar objects that require similar handling. It favors composition over inheritance and follows the open/close principle.

After finishing this chapter, they will be able to understand the working of the composite design pattern and how it can be used to model real-world *part–whole* relationships, making your applications easier to reason about, extend, and maintain.

Composite design pattern

The composite design pattern is a structural design pattern that allows for simplifying the handling of hierarchical data by providing a common interface for both single elements and whole collections.

When building software, it is common to come across situations where items can contain other items of the same kind. For example, a product category might contain subcategories, a shape in a graphics editor might be made of smaller shapes, and a company's organizational chart is built from employees who may also be managers of other employees. These structures look and behave like trees.

The composite design pattern provides a clean way to represent and work with these tree structures. It lets you define **single elements** and **groups of elements** in a way that they can be handled through the **same interface**. This means your code can remain simple, and you do not need special cases for whether you are dealing with one object or a collection. The pattern encourages flexibility and makes it much easier to add new types of elements in the future without modifying existing logic.

Figure 13.2 exhibits a real-life example depicting organizational hierarchy, where a team can be made up of managers, and each manager can have either other managers or **individual contributors** (**ICs**) reporting to them:

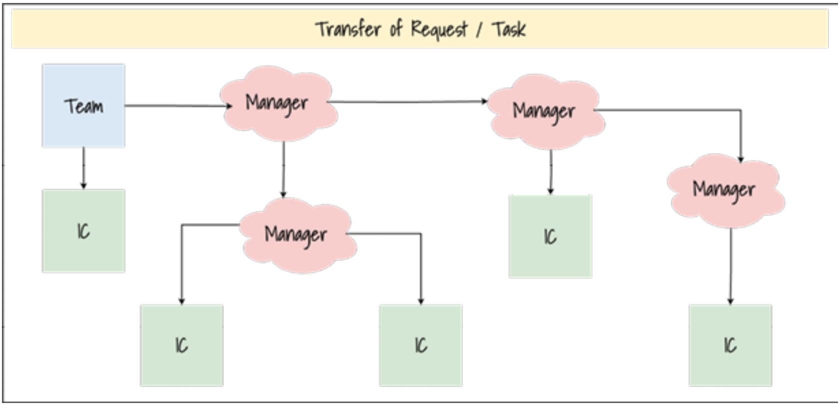


Figure 13.2*: Representation of organizational structure*

In the composite design pattern, this structure is recognized as a **tree**, where:

- **Team**: Represents the root composite.
- **Managers**: Represent composite nodes (they manage others).
- **ICs**: Represent leaf nodes (they do not manage others).

Both managers and ICs share a common interface, and they can receive and handle a request or task.

This pattern promotes writing code that does not need to care whether it is dealing with one object or a collection of objects, which helps reduce complexity, encourages consistent behavior across components, and makes it easier to add new types of objects without changing existing logic.

Figure 19.3 exhibits the architectural diagram of the composite design pattern. It displays the connection between the **Component** interface and the two distinct derived component types. The **Complex Component** contains an object of type **Component** that allows us to create a hierarchy of objects that can be treated alike.

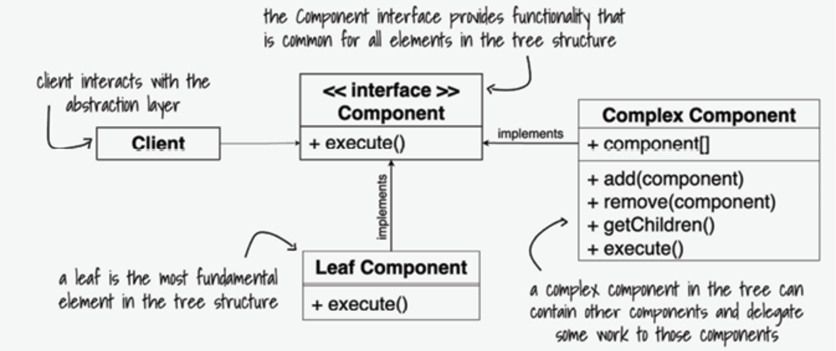

the Component interface provides functionality that is common for all elements in the tree structure
client interacts with the abstraction layer
<< interface >>
Component
+ execute()
Client
implements
Complex Component
+ component[]
+ add(component)
+ remove(component)
+ getChildren()
+ execute()
implements
Leaf Component
+ execute()
a leaf is the most fundamental element in the tree structure
a complex component in the tree can contain other components and delegate some work to those components

Figure 19.3*: Composite design pattern*

The component design pattern follows the principle of uniformity and allows the user to treat distinct but similar objects uniformly. In the preceding figure, both the complex and the leaf component can be imagined having the same set of fields that define its state, allowing for similar treatment.

Implementing the composite pattern

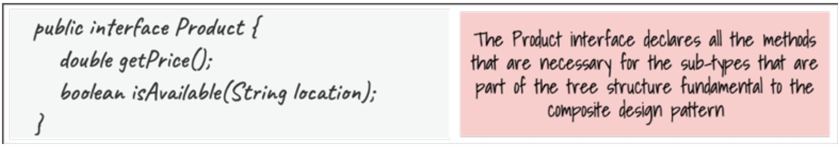
The example that we will discuss is this chapter takes inspiration from objects commonly used in e-commerce and food tech applications. This example will explain the working of the component design pattern and will help to understand the basics of utilizing hierarchical relationships in object construction.

We often order food online or purchase from our nearest eatery using kiosks. The menus often display food items that can be purchased individually or in the form of meals. These meals are often just a group of the same items, but with a discounted price on them. These meals not only save money but also save time, and therefore are widely ordered across these platforms.

This approach gives out an impression of having two distinct types of food items, but they are not much different. Since these meals are just a pack of the individual items, it makes a lot of sense to treat both uniformly. We will build upon this idea and implement a small working example showcasing the benefit of using the composite design pattern whenever such a use case arises.

Product interface

The composite design pattern utilizes abstraction and polymorphism to build hierarchical relations between objects. To make use of composition, all the related objects must implement the **Product** interface. *Figure 13.4* exhibits the **Product** interface that defines the behavior that must be implemented by the derived types:



```
public interface Product {
    double getPrice();
    boolean isAvailable(String location);
}
```

The Product interface declares all the methods that are necessary for the sub-types that are part of the tree structure fundamental to the composite design pattern

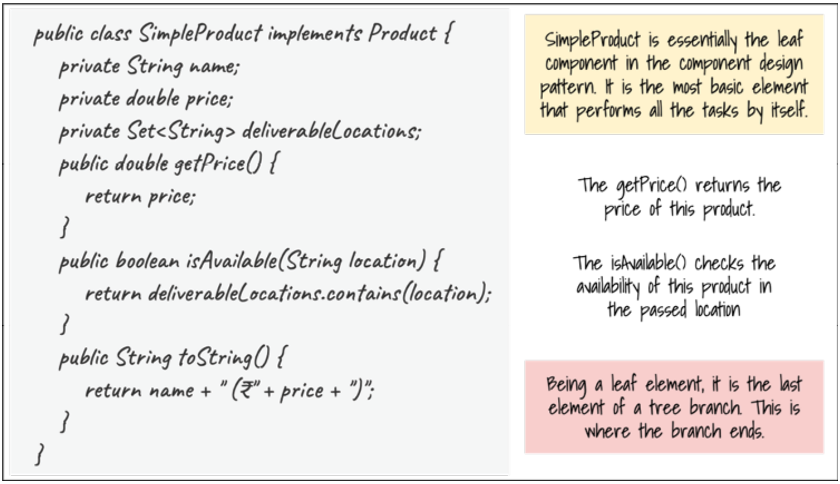
Figure 13.4: *Product interface*

In the preceding figure, the interface exposes two methods that must be implemented by each of the component types. The implementation itself is of utmost importance here, as we will uncover in the following sections.

Derived types

Uniform treatment of the underlying components is the core idea behind the component design pattern. While the interface mandates this functionality, it is still very much dependent on the derived types to provide a correct implementation to ensure uniform treatment of the components.

The **getPrice** and **isAvailable** methods from the interface discussed in the previous section must be properly implemented for that very purpose. *Figure 13.5* exhibits the implementation of one of the components, a **SimpleProduct**:



```
public class SimpleProduct implements Product {
    private String name;
    private double price;
    private Set<String> deliverableLocations;
    public double getPrice() {
        return price;
    }
    public boolean isAvailable(String location) {
        return deliverableLocations.contains(location);
    }
    public String toString() {
        return name + " (₹" + price + ")";
    }
}
```

SimpleProduct is essentially the leaf component in the component design pattern. It is the most basic element that performs all the tasks by itself.

The getPrice() returns the price of this product.

The isAvailable() checks the availability of this product in the passed location

Being a leaf element, it is the last element of a tree branch. This is where the branch ends.

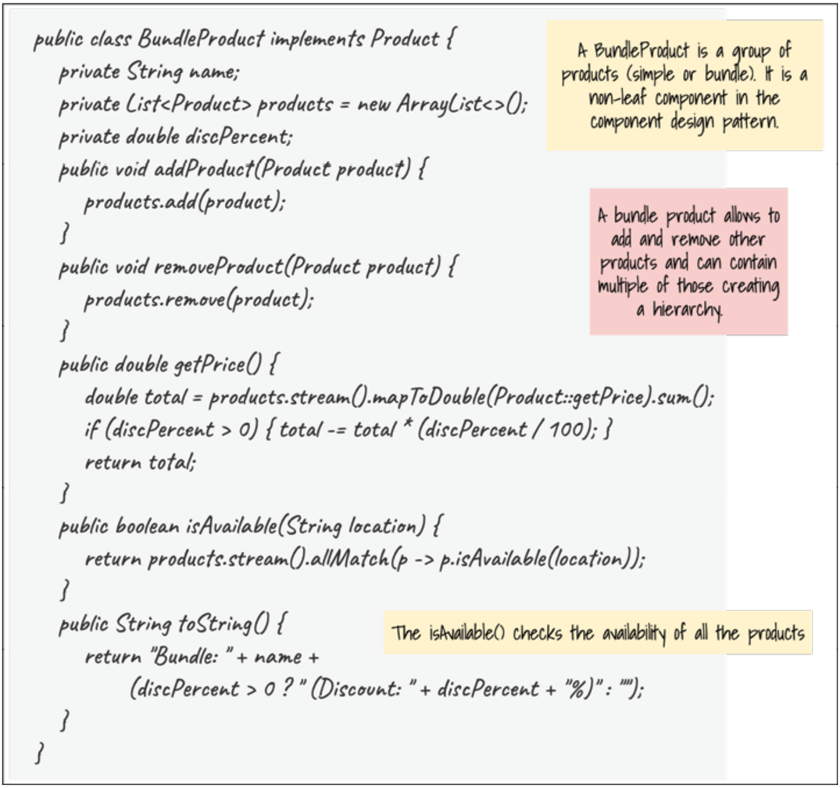
***Figure 13.5**: Component of SimpleProduct*

It is the leaf element in our example that performs all the tasks by itself and does not delegate to other components. It serves as the last element of every branch of the hierarchical structure, which means that no further elements are accessible further in that branch.

As stated earlier, a simple product is an individual product that can be purchased separately from an e-commerce application. The **SimpleProduct** implementation, as shown in *Figure 13.5*, provides definitions to the methods exposed by the **Product** interface. These definitions interact with the state of the object. Being a leaf element, it does not interact with other elements of the **Product** type and acts as the last element in that branch of hierarchy.

The complex component, on the other hand, is the most interesting element in the component design pattern. It is the property of this element to control the formation of hierarchy and to allow access to the lower-level elements. Each of these lower-level elements in turn can be a simple or complex element themselves.

Figure 13.6 exhibits the implementation of **BundleProduct**, the complex component in our example. A bundle product is more like a meal, a pack of individual products that are sold in a group. They have their own price and can contain both individual and bundle products.



```
public class BundleProduct implements Product {
    private String name;
    private List<Product> products = new ArrayList<>();
    private double discPercent;
    public void addProduct(Product product) {
        products.add(product);
    }
    public void removeProduct(Product product) {
        products.remove(product);
    }
    public double getPrice() {
        double total = products.stream().mapToDouble(Product::getPrice).sum();
        if (discPercent > 0) { total -= total * (discPercent / 100); }
        return total;
    }
    public boolean isAvailable(String location) {
        return products.stream().allMatch(p -> p.isAvailable(location));
    }
    public String toString() {
        return "Bundle: " + name +
            (discPercent > 0 ? " (Discount: " + discPercent + "%)" : "");
    }
}
```

A BundleProduct is a group of products (simple or bundle). It is a non-leaf component in the component design pattern.

A bundle product allows to add and remove other products and can contain multiple of those creating a hierarchy

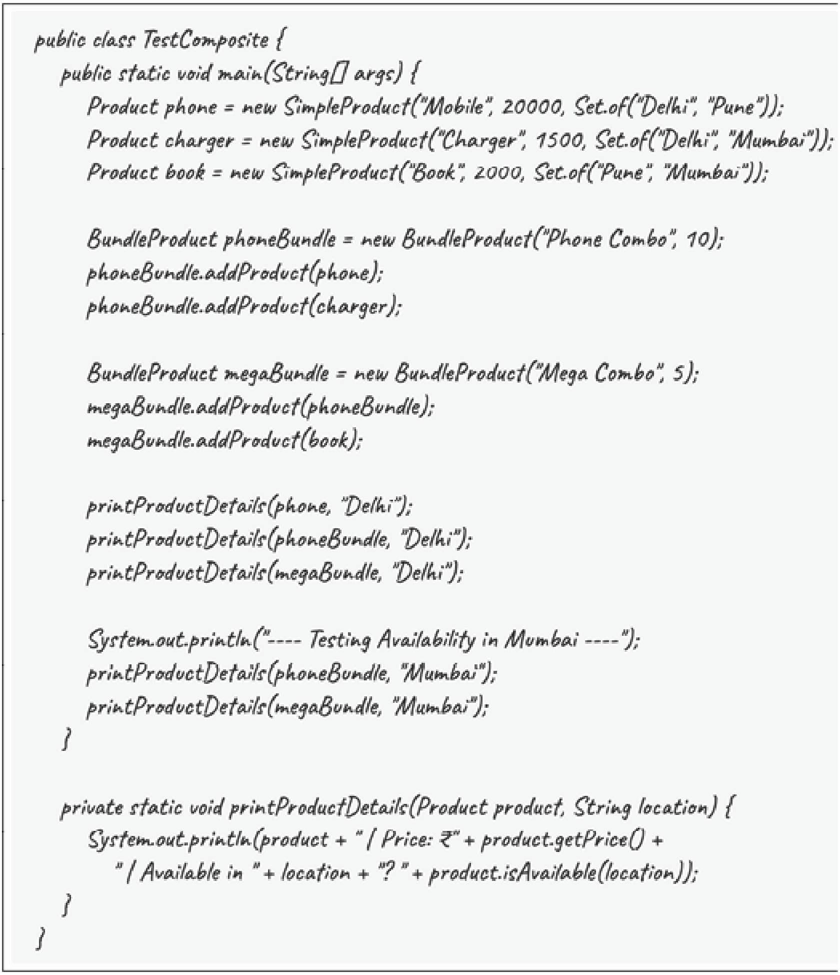
The isAvailable() checks the availability of all the products

Figure 13.6*: Component of SimpleProduct*

To calculate the price and availability of the bundle, it is necessary to ensure that the overridden methods in the **BundleProduct** implementation include access to all the products below it in the hierarchy. For example, if one must check for the price of a bundle product, the implementation must recursively iterate over all the other components that form the list. Having a reference to a list of components allows the complex component to control the hierarchy.

Testing the composite design pattern

We have written a test class to verify the working of the composite design pattern. Let us execute the same as shown in *Figure 13.7*:

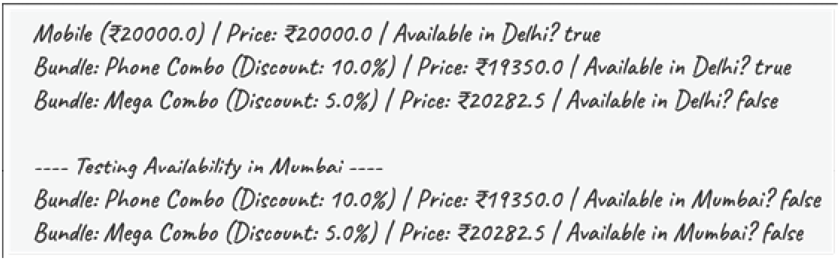


```
public class TestComposite {
    public static void main(String[] args) {
        Product phone = new SimpleProduct("Mobile", 20000, Set.of("Delhi", "Pune"));
        Product charger = new SimpleProduct("Charger", 1500, Set.of("Delhi", "Mumbai"));
        Product book = new SimpleProduct("Book", 2000, Set.of("Pune", "Mumbai"));

        BundleProduct phoneBundle = new BundleProduct("Phone Combo", 10);
        phoneBundle.addProduct(phone);
        phoneBundle.addProduct(charger);

        BundleProduct megaBundle = new BundleProduct("Mega Combo", 5);
        megaBundle.addProduct(phoneBundle);
        megaBundle.addProduct(book);

        printProductDetails(phone, "Delhi");
        printProductDetails(phoneBundle, "Delhi");
        printProductDetails(megaBundle, "Delhi");

        System.out.println("---- Testing Availability in Mumbai ----");
        printProductDetails(phoneBundle, "Mumbai");
        printProductDetails(megaBundle, "Mumbai");
    }

    private static void printProductDetails(Product product, String location) {
        System.out.println(product + " / Price: ₹" + product.getPrice() +
            " / Available in " + location + "? " + product.isAvailable(location));
    }
}
```

***Figure 13.7**: Testing the pipeline*

In *Figure 13.7*, the test implementation first creates some simple products and then uses those products to create new bundle products. The intent of this test is to verify that the implementation of the bundle product ensures that it interacts with all the simple products that are part of that bundle while checking for the price and availability of the bundle itself.

Figure 13.8 exhibits the output of the test code from *Figure 13.7*:



```
Mobile (₹20000.0) / Price: ₹20000.0 / Available in Delhi? true
Bundle: Phone Combo (Discount: 10.0%) / Price: ₹19350.0 / Available in Delhi? true
Bundle: Mega Combo (Discount: 5.0%) / Price: ₹20282.5 / Available in Delhi? false

---- Testing Availability in Mumbai ----
Bundle: Phone Combo (Discount: 10.0%) / Price: ₹19350.0 / Available in Mumbai? false
Bundle: Mega Combo (Discount: 5.0%) / Price: ₹20282.5 / Available in Mumbai? false
```

***Figure 13.8**: Test result*

Summary

The composite design pattern aims to represent complex object hierarchies by organizing them into tree-like structures composed of simple and composite elements. It utilizes abstraction and promotes uniform treatment of individual objects and groups of objects through a shared interface. The structure itself is extensible and allows components to be added or removed at any level, be it a leaf node, a branch, or the root of the hierarchy.

In *Figure 13.9*, the composite design pattern is reimaged with the help of the classes that we have used in our example:

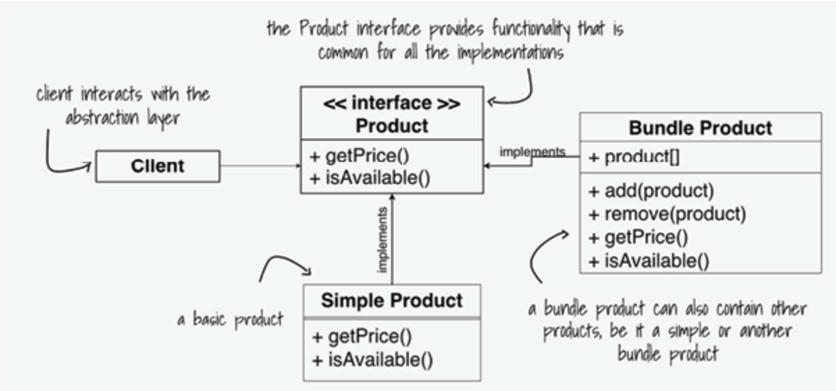

the Product interface provides functionality that is common for all the implementations
client interacts with the abstraction layer
Client
<< interface >>
Product
+ getPrice()
+ isAvailable()
implements
Bundle Product
+ product[]
+ add(product)
+ remove(product)
+ getPrice()
+ isAvailable()
implements
a basic product
Simple Product
+ getPrice()
+ isAvailable()
a bundle product can also contain other products, be it a simple or another bundle product

***Figure 13.9**: Pipeline design pattern—defined*

Individual components can be reused across different compositions and combined in various configurations to exhibit different structural behaviors. The composite can be implemented in different ways based on whether the components perform operations independently or delegate them recursively to their child elements.

Advantages and drawbacks

It is time to make a note of some of the benefits and drawbacks of the composite design pattern. Let us start with the benefits first followed by drawbacks:

- By organizing objects into hierarchical structures, the system becomes easier to manage and extend.
- Uniform treatment of individual objects and compositions simplifies client interaction and reduces code complexity.
- Reusability is promoted by allowing components to be used across different composite structures.
- It promotes flexibility and allows altering the hierarchy's behavior by adding or removing components without modifying existing code.

The drawbacks of the pipeline design pattern are:

- Overhead may increase due to the need to maintain a recursive structure and shared interface.
- Debugging and maintaining deep hierarchies can be

challenging due to the recursive nature of the pattern.

- The uniform interface might limit specialized behavior for certain components.

Usage

Some of the scenarios relevant to the pipeline design pattern are:

- It can be used to represent part-whole hierarchies where individual objects and groups of objects need to be treated uniformly.
- When there is a need to work with complex structures while keeping client code simple, the composite pattern provides a unified way to access all elements.
- It is suitable when objects need to be dynamically composed or decomposed without affecting the overall system structure.
- When operations must apply uniformly across simple and composite elements, the composite pattern offers an elegant solution.

Conclusion

The composite design pattern aims to represent part-whole hierarchies by allowing individual objects and compositions of objects to be treated uniformly. It defines a common abstraction for both simple and complex elements, enabling clients to interact with them through the same interface. This promotes flexibility and simplifies code that works with object hierarchies.

The pattern supports recursive composition, where complex objects are formed by combining simpler ones, which can themselves contain further components. The structure is easily extensible, allowing new component types to be added without modifying existing code.

By promoting loose coupling between clients and object compositions, the composite pattern enables dynamic tree structures where leaf nodes and composite nodes can be used interchangeably to exhibit consistent behavior.

In the next chapter, we will build an understanding of the behavioral design patterns, exploring how they define and manage interactions between objects. We will start with the template method pattern, learning how it enables controlled variation in behavior through the elegant principles of object-oriented programming.

Questions

In this section, we will sharpen our brains by indulging in some fun brain activities. Let us get started.

Find the odd one out

1. Composite design pattern:
    a. Represents part-whole hierarchies
    b. It is a structural design pattern
    c. Allows treating objects uniformly
    d. Defines the object creation process
2. Components in a composite structure:
    a. Share a common interface
    b. Can contain other components
    c. Must always be leaf nodes
    d. Can be treated uniformly by clients
3. Composite pattern promotes:
    a. Flexibility and extensibility
    b. Uniform access to objects and groups
    c. Recursive composition of objects
    d. Tight coupling between client and structure

Select two correct answers

1. Which of the following statements correctly describes the composite design pattern?
    a. It allows clients to treat individual objects and compositions of objects uniformly

   b. It is primarily used to represent hierarchical tree structures
   c. It requires leaf nodes to expose a different interface than composite nodes
   d. It prevents recursive structures from being formed

2. In a typical implementation of the composite design pattern, which responsibilities belong to the composite class?
   a. Maintaining a collection of child components
   b. Delegating operations to child components
   c. Representing terminal objects that cannot have children
   d. Restricting clients from invoking operations on leaf nodes

Answers

This section provides the answers to the questions given.

Find the odd one out

1. Defines the object creation process
2. Must always be leaf nodes
3. Tight coupling between client and structure

Select two correct answers

1. It allows clients to treat individual objects and compositions of objects uniformly, it is primarily used to represent hierarchical tree structures
2. Maintaining a collection of child components, Delegating operations to child components

CHAPTER 14

Template

Introduction

As developers, we often find so many similar algorithms being written time and again, with minimal changes. These changes often apply only to a part of the complete algorithm, and we end up duplicating a lot of code. It requires a good amount of effort to rework and test most of that algorithm, again.

The algorithms should be dynamic and reusable, made of multiple smaller chunks of code, and should provide placeholders that can be filled in with appropriate method definitions to support varying behavior. *Figure 14.1* demonstrates an algorithm that is an amalgamation of multiple methods:

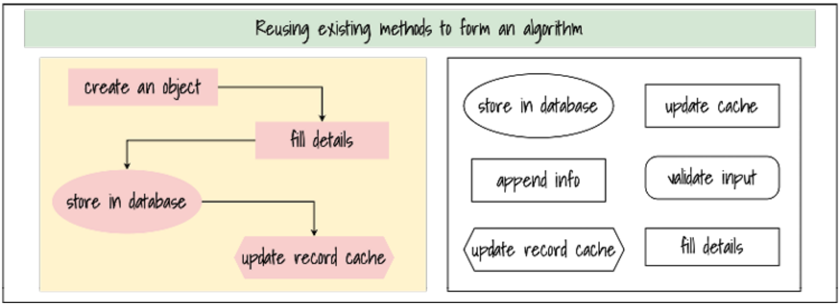

Reusing existing methods to form an algorithm
create an object
fill details
store in database
update record cache
store in database
update cache
append info
validate input
update record cache
fill details

Figure 14.1: Reusing methods

The left side of the figure describes the flow of the algorithm, and the right side represents the methods that form that algorithm.

Although it is a good practice to break down large methods into multiple smaller ones that perform specific tasks so that they can be reused, they might not always be suited due to the variation in the method signature. Such methods might require a change in signature or be wrapped within another method. The order and sequence in which these methods should be called is also of importance.

What if we construct abstract methods in place of concrete ones and allow them to be overridden or replaced according to the desired behavior?

Structure

This chapter covers the following topics:

- Template
- Template method design pattern
- Building a template method
- Summary
- Advantages and drawbacks
- Usage
- Conclusion

Objectives

By the end of this chapter, the readers will be able to understand the dynamic behavior of algorithms and to learn how to apply polymorphism to bring that dynamism to life. They will learn about the template method design pattern and how it can be utilized to alter the behavior of an algorithm at runtime.

The chapter will also walk them through the implementation details of the template method design pattern, the benefits it provides, and a few drawbacks, along with some of the places where this design pattern can be used.

Template

Have you ever received an invitation to an Indian marriage? Wonder why they look so similar and often have the same text. The format of the dates, the way the names are mentioned, and the mantras that form an indistinguishable part of the invitation look so much alike. This is because the wedding cards usually follow a set format that has been in existence since time immemorial. *Figure 14.2* demonstrates an algorithm in the form of a template:

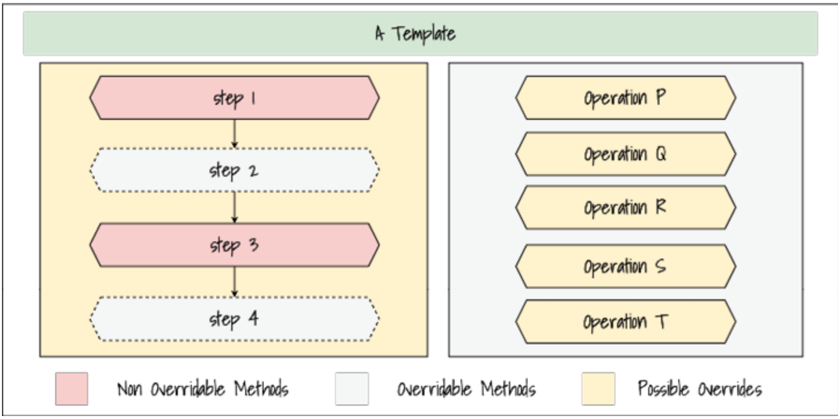

A Template
step 1
step 2
step 3
step 4
Operation P
Operation Q
Operation R
Operation S
Operation T
Non Overridable Methods
Overridable Methods
Possible Overrides

***Figure 14.2**: A template*

The preceding algorithm is divided into four steps; some of those steps are fixed, while others could be overridden to change the default behavior.

Let us get straight to the point. There is a template that is being followed for preparing the wedding cards, a representation that we are accustomed to. It also gives fewer headaches to those who prepare and print these wedding cards. So, our next big question is, *how can we define a template?*

In computing, a template is a skeleton, a kind of model. It defines a sequence of operations that is organized in an order. It could also be understood as an arrangement of steps that are taken to process an algorithm. These steps could be abstract or concrete, depending on the arrangement of steps in a template.

Template method design pattern

The template method design pattern is a behavioral design pattern that allows for the creation of a skeleton of operations. It provides the high-level steps of an algorithm. The steps are in the form of methods that can be overridden to support dynamic behavior without making any change in the overall structure. *Figure 14.3* demonstrates the class diagram of the template method design pattern:

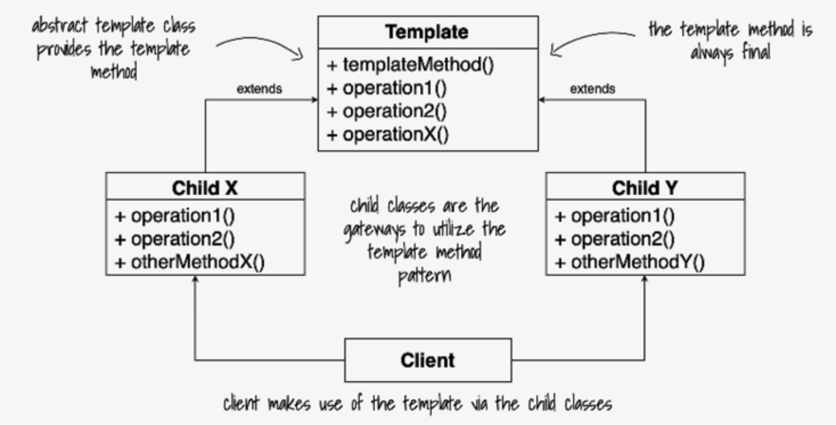

abstract template class provides the template method
Template
+ templateMethod()
+ operation1()
+ operation2()
+ operationX()
the template method is always final
extends
extends
Child X
+ operation1()
+ operation2()
+ otherMethodX()
child classes are the gateways to utilize the template method pattern
Child Y
+ operation1()
+ operation2()
+ otherMethodY()
Client
client makes use of the template via the child classes

***Figure 14.3**: Template method design pattern*

The class that exposes the template method is called the parent class. The template method preserves the sequence of the algorithm and is therefore marked final to restrict it from being overridden. It does not have any definition of its own; it merely acts as an organizer that operates the flow of the algorithm by delegating responsibilities to other methods.

The parent class has two types of methods:

- The final ones that cannot be overridden.
- The abstract ones that must be overridden.

The fixed or the final ones are often referred to as default implementations or the helper methods. The abstract ones are also called **placeholders** or **hooks**.

Figure 14.4 demonstrates the structure of a template method and how child classes can override it to alter the definition of an algorithm:

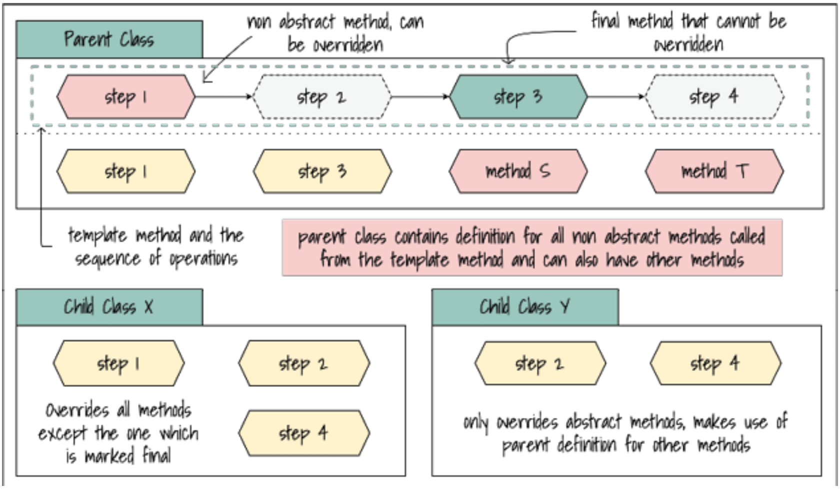

Parent Class
non abstract method, can be overridden
final method that cannot be overridden
step 1
step 2
step 3
step 4
step 1
step 3
method S
method T
template method and the sequence of operations
parent class contains definition for all non abstract methods called from the template method and can also have other methods
Child Class X
step 1
step 2
step 4
Overrides all methods except the one which is marked final
Child Class Y
step 2
step 4
only overrides abstract methods, makes use of parent definition for other methods

***Figure 14.4**: Template method and parent class*

The parent class usually provides helper methods for the common, generic kind of implementation that is required in most of the variations of the algorithm. The child class is responsible for implementing the rest of the skeleton by providing definitions to the abstract methods defined by the parent.

The dynamic behavior that is at the core of the template method pattern is made possible due to inheritance and polymorphism, one of the four fundamental legs of object-oriented programming. It is only through a child object that the template method can be utilized. The template method calls all the other methods that form the sequence of steps of the algorithm it exposes.

While some of the method definitions are provided by the parent itself, definitions for all the abstract ones are provided by the child. It is at this moment that polymorphism comes into action, and the child implementation is called for all the hook or placeholder entries in the template method pattern.

Here is how it works:

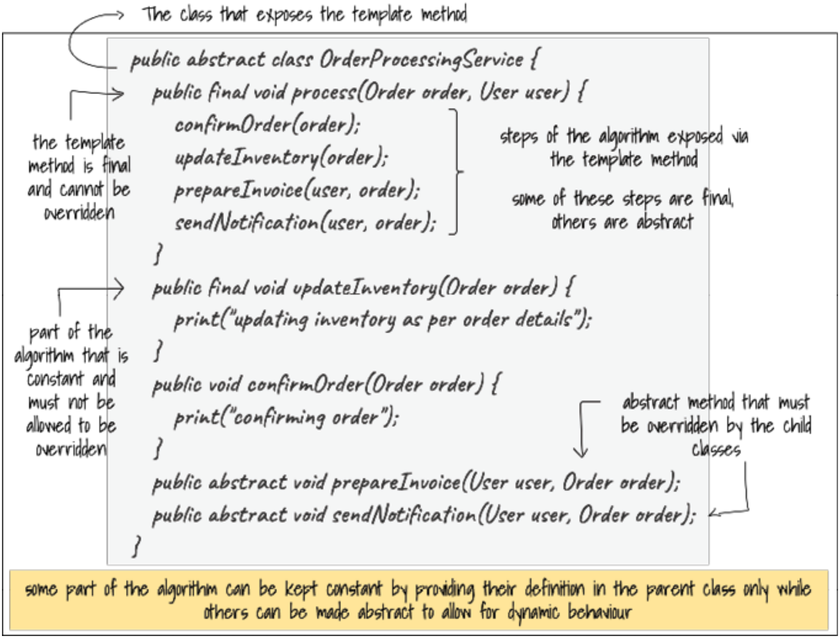
- Instantiate an object of the child class.
- Call the template method on the child class object.
- The template method calls other methods in a defined sequence.
- Each time when a call is made, and if the called method is overridden by the child, the child implementation is called; otherwise, the parent implementation is used.

Building a template method

The example that we will walk you through makes use of an abstract class that exposes the template method and a few of its subclasses. The subclasses override the abstract methods to complete the algorithm that is exposed as a template method. We will learn about implementing the template method and how it helps to shape the behavior of an algorithm while keeping its structure intact. Let us first look at the abstract class that provides the template method and forms the skeleton of our algorithm. Later, we will discuss the implementations that enable dynamic behavior.

Template method

The template method provides the skeleton of an algorithm. It lays out a sequence of steps that are executed in sequence. The provided steps are in the form of methods that are defined by the parent class or its subclasses. In *Figure 14.5*, the process method serves as the template method:



```
public abstract class OrderProcessingService {
    public final void process(Order order, User user) {
        confirmOrder(order);
        updateInventory(order);
        prepareInvoice(user, order);
        sendNotification(user, order);
    }
    public final void updateInventory(Order order) {
        print("updating inventory as per order details");
    }
    public void confirmOrder(Order order) {
        print("confirming order");
    }
    public abstract void prepareInvoice(User user, Order order);
    public abstract void sendNotification(User user, Order order);
}
```

The class that exposes the template method
the template method is final and cannot be overridden
steps of the algorithm exposed via the template method
some of these steps are final, others are abstract
part of the algorithm that is constant and must not be allowed to be overridden
abstract method that must be overridden by the child classes
some part of the algorithm can be kept constant by providing their definition in the parent class only while others can be made abstract to allow for dynamic behaviour

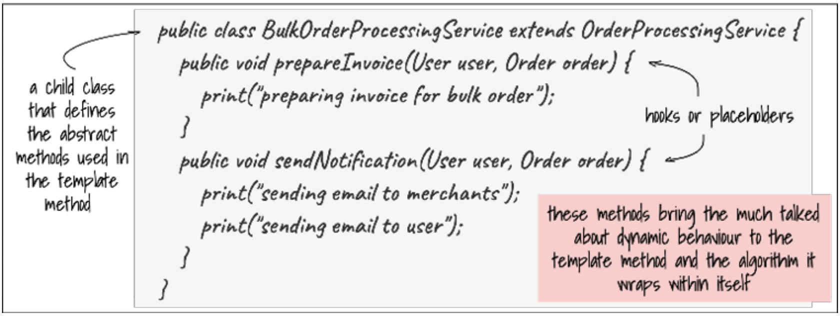
***Figure 14.5**: Template method*

It contains calls to four different methods that complete the order processing after a successful payment. The methods that form a part of the template method can be final as well as abstract depending on what task they perform and whether that task is invariable or variable.

In every algorithm that can be made generic to work for many different use cases, some part of it can be common enough to not need any variation, or it is a must to perform in only one way. Such parts of the algorithm are defined in the template using methods that are declared as final. It ensures that they are not accidentally overridden. For other parts of the algorithm, the parent class could either provide default method implementations or abstract methods. The non-final methods facilitate the dynamic behavior of the algorithm, as the methods that are abstract in the parent class must be overridden by the subclasses.

Varying implementations

The abstract methods that we talked about need a definition so that the algorithm exposed using the template method can be provided with a proper, overall definition. The child classes control the behavior of the algorithm by providing definitions to those methods. In *Figure 14.6*, the child class has overridden both the abstract methods:



```
public class BulkOrderProcessingService extends OrderProcessingService {
    public void prepareInvoice(User user, Order order) {
        print("preparing invoice for bulk order");
    }
    public void sendNotification(User user, Order order) {
        print("sending email to merchants");
        print("sending email to user");
    }
}
```

a child class that defines the abstract methods used in the template method

hooks or placeholders

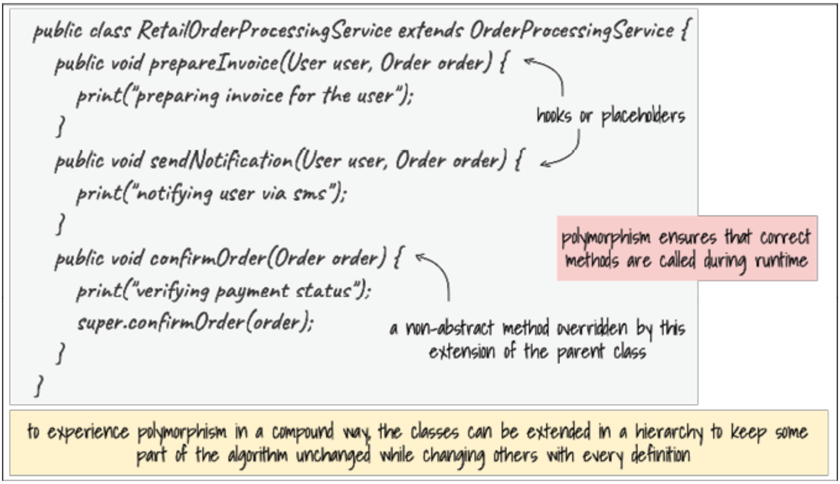
these methods bring the much talked about dynamic behaviour to the template method and the algorithm it wraps within itself

***Figure 14.6**: Implementation I*

The definitions provided by these implementations shape the behavior of the algorithm. Polymorphism plays a big role in ascertaining which behavior will be selected during runtime based on the type of object the template method is called on. For example, when the template method is called on an object of the type described in *Figure 14.6*, the notification is sent to the merchants as well as the user.

These abstract methods are placed in the template as hooks or placeholders and are supposed to be overridden by the implementations. While the abstract methods should mandatorily be overridden, the methods with default implementation can also be overridden if required.

In *Figure 14.7*, the child class not only overrides the abstract methods but also overrides a method that already has a definition in the parent class:



```
public class RetailOrderProcessingService extends OrderProcessingService {
   public void prepareInvoice(User user, Order order) {
      print("preparing invoice for the user");
   }
   public void sendNotification(User user, Order order) {
      print("notifying user via sms");
   }
   public void confirmOrder(Order order) {
      print("verifying payment status");
      super.confirmOrder(order);
   }
}
```

hooks or placeholders

polymorphism ensures that correct methods are called during runtime

a non-abstract method overridden by this extension of the parent class

to experience polymorphism in a compound way, the classes can be extended in a hierarchy to keep some part of the algorithm unchanged while changing others with every definition

***Figure 14.7**: Implementation II*

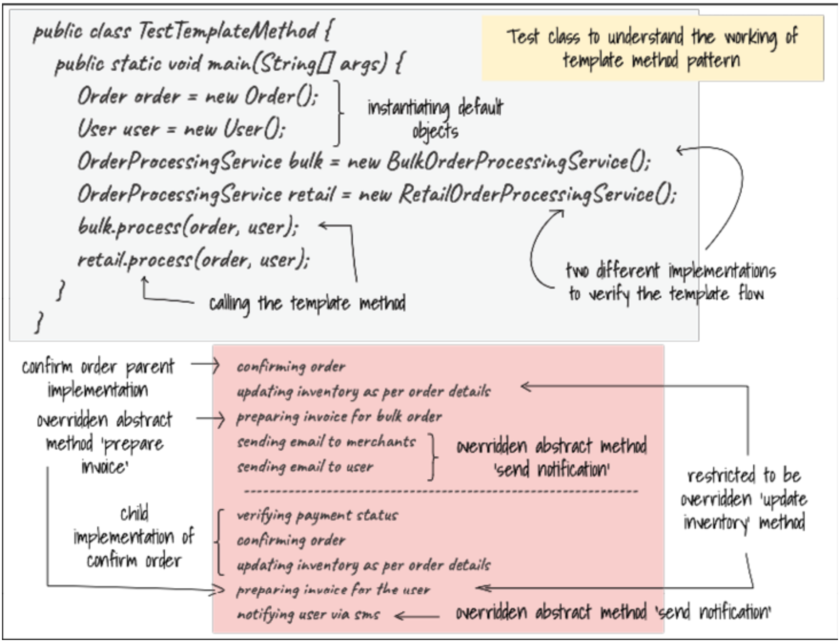
This scenario demonstrates that some methods can have a default implementation that may or may not be overridden based on the use case. While some child classes may end up providing a different implementation to those methods, others can just make use of the default one.

These implementations are only for understanding the concepts and might not be relevant for actual business logic. You are advised to exercise caution and only use it for learning purposes.

Testing the template method pattern

We have written a test class to verify the working of the template methods pattern. Let us execute the same.

In *Figure 14.8*, we first instantiate objects of both implementations of the parent class and then call the template method on both:



```
public class TestTemplateMethod {
  public static void main(String[] args) {
    Order order = new Order();
    User user = new User();
    OrderProcessingService bulk = new BulkOrderProcessingService();
    OrderProcessingService retail = new RetailOrderProcessingService();
    bulk.process(order, user);
    retail.process(order, user);
  }
}
```

```
confirming order
updating inventory as per order details
preparing invoice for bulk order
sending email to merchants
sending email to user
------------------------------------------------------------
verifying payment status
confirming order
updating inventory as per order details
preparing invoice for the user
notifying user via sms
```

Test class to understand the working of template method pattern
instantiating default objects
two different implementations to verify the template flow
calling the template method
confirm order parent implementation
overridden abstract method 'prepare invoice'
overridden abstract method 'send notification'
restricted to be overridden 'update inventory' method
child implementation of confirm order
overridden abstract method 'send notification'

Figure 14.8: *Test template method*

The difference in the output confirms two different behaviors achieved using the template method pattern.

Summary

The template method design pattern allows for defining a generic algorithm, which can provide different behaviors by overriding the methods that are a part of the template method. The methods can be abstract or final, depending on what part of the algorithm must be kept uniform and what should be kept variable.

Figure 14.9 demonstrates the class diagram of the template method design pattern in the context of the example used in this chapter:

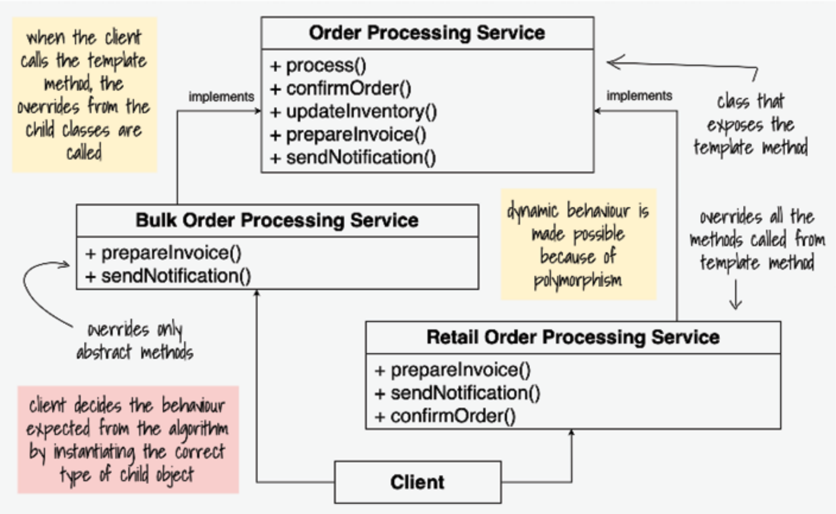

when the client calls the template method, the overrides from the child classes are called
Order Processing Service
+ process()
+ confirmOrder()
+ updateInventory()
+ prepareInvoice()
+ sendNotification()
implements
implements
class that exposes the template method
Bulk Order Processing Service
+ prepareInvoice()
+ sendNotification()
dynamic behaviour is made possible because of polymorphism
overrides all the methods called from template method
overrides only abstract methods
Retail Order Processing Service
+ prepareInvoice()
+ sendNotification()
+ confirmOrder()
client decides the behaviour expected from the algorithm by instantiating the correct type of child object
Client

***Figure 14.9**: Template method pattern—defined*

The client is responsible for deciding the behavior of the algorithm by selecting the correct child implementation. Polymorphism then ensures that the right set of methods is called when the template method is invoked on the object.

Advantages and drawbacks

It is time to make note of some of the benefits and drawbacks of the template method design pattern. Let us start with the benefits first followed by the drawbacks:

- Easy to introduce new behavior to an existing algorithm.
- Promotes code reusability.
- Avoids code duplication by implementing the common steps.

The drawbacks of the template method design pattern are as follows:

- The template has a predefined sequence of steps; any change in the ordering will require a new template.
- Having multiple abstract steps could be hard to manage.
- The sequence of steps should be known before the pattern is applied.

Usage

The following list outlines some of the scenarios relevant to creating a template method:

- When multiple outcomes are possible with a slight change in the algorithm.
- For writing frameworks where only the necessary steps are exposed to the implementing classes.

Conclusion

The template method design pattern is a widely used behavioral design pattern that can be utilized to expose a sequence of steps to form an algorithm. These steps can be defined in multiple ways by various implementations to provide a wide range of behavior.

In the next chapter, we will learn about the observer design pattern and how it can be used to notify state changes of an object to its observers.

Questions

In this section, we will sharpen our brains by indulging in some fun brain activities. Let us get started.

Find the odd one out

1. Template method, final, abstract, sequence of steps
2. Child implementation, abstract, non-overridable, overridable
3. Parent provides default implementation, child provides specific implementation, final template method, abstract template method
4. The template method design pattern
    a. Promotes code reusability
    b. Provides a sequence of steps to perform some task
    c. Reduces readability
    d. Relies on polymorphism

5. The template method design pattern
    a. Can be used to reduce code duplication
    b. Allows multiple behaviors with minor changes
    c. Can have final methods
    d. Is a structural design pattern
6. The template method
    a. Is final
    b. Provides a skeleton
    c. Is abstract
    d. Can call abstract and non-abstract methods

Select two correct answers

1. The template method design pattern
    a. Promotes code reusability
    b. Can be used without polymorphism
    c. Cannot be used with inheritance
    d. Reduces code duplication
2. The template method
    a. Allows for multiple behaviors
    b. Can be used to drive behavior using conditional statements
    c. Is exposed by the child classes
    d. Contains only abstract methods
3. What goes together?
    a. Polymorphism, template method, abstract, final
    b. Dynamic behavior, behavioral, template method
    c. Structural, template method pattern, sequence of steps
    d. Polymorphism, inheritance, template, skeleton

Answers

This section provides the answers to the given questions.

Find the odd one out

1. Final
2. Non-overridable
3. Abstract template method
4. Reduces readability
5. Is a structural design pattern
6. Is abstract

Select two correct answers

1. Promotes code reusability, reduces code duplication
2. Allows for multiple behaviors, can be used to drive behavior using conditional statements
3. Dynamic behavior, behavioral, template method, and polymorphism, inheritance, template, skeleton

Join our Discord space

Join our Discord workspace for latest updates, offers, tech happenings around the world, new releases, and sessions with the authors:

https://discord.bpbonline.com

CHAPTER 15

Keep a Close Eye

Introduction

Transferring information in a controlled manner is a common use case among systems that coexist and cooperate. The information that is being transferred could be an integer, a text value, or maybe an object that stores complex information. It largely depends on the use case and its implementation. If there are only two systems involved, then it becomes a straightforward solution where one system is the sender of the information, and the other is the receiver of that information but with a large set of receivers, the same implementation is not enough; it requires changes to accommodate multiple receivers that are interested in the same information.

Systems that rely on each other for updates often prefer some sort of asynchronous communication. This helps to keep the resources in check and prevents them from being used up for unnecessary waiting periods. A subscribe and notify mechanism is one such way that can be used to transfer information from the sender to the receiver. The sender often keeps a list of all the receivers who are interested in the information and updates them whenever there is a change.

In this chapter, we will discuss the observer design pattern and how it can be utilized to broadcast information to all the interested subscribers.

Structure

This chapter covers the following topics:

- Observer design pattern
- Building observer pattern
- Varying implementations
- Pull mechanism
- Summary
- Advantages and drawbacks
- Usage

Objectives

By the end of this chapter, the readers will understand the mechanism of subscription-based information transfer. When some data or a change in data is of relevance to multiple systems, transferring it to all of them is a challenge. The observer design pattern helps us to address that challenge in a way that is both organized and asynchronous in nature.

The chapter will walk them through the implementation details of the observer design pattern, the benefits it provides, and a few drawbacks, along with some of the places where this design pattern can be used.

Observer design pattern

The observer design pattern is a behavioral design pattern that allows for defining a subscribe and notify-based mechanism for transferring information. The two kinds of objects that form the basis of the observer design pattern are called the subject and the observer. The subject is the point of interest, and the observers are interested in it. In short, the subject is what is being observed by the observers. The subject allows itself to be observed and is also referred to as an observable.

The observer design pattern describes a one-to-many relationship between objects where multiple observers observe a single subject.

These observers subscribe to the subject and are notified of any change in its state. In *Figure 15.1*, multiple observers are depicted as registering themselves with a subject:

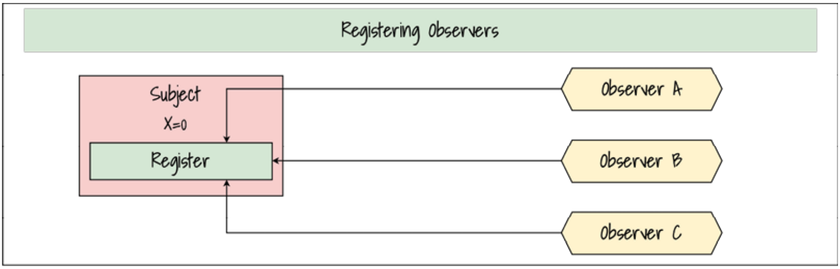

Registering Observers
Subject
X=0
Register
Observer A
Observer B
Observer C

***Figure 15.1**: Registering servers*

An observer sends a request to the subject with a reference to itself to receive updates whenever the subject's state changes. The same process must be followed every time an observer is interested in a subject. In *Figure 15.1*, the value of X, which forms the state of the Subject, is the point of interest for all the registered observers.

In *Figure 15.2*, the subject is depicted as notifying all the active observers about the change in the value of X, which is now 5:

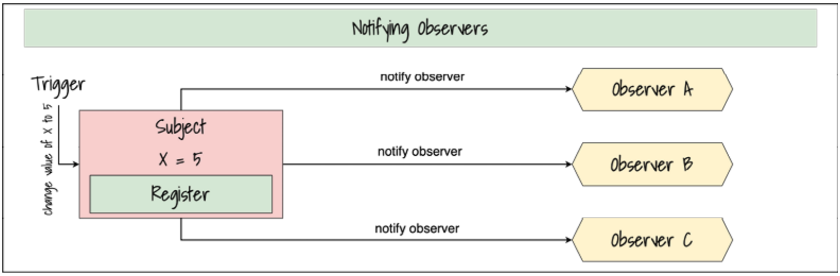

Notifying Observers
Trigger
change value of X to 5
Subject
X = 5
Register
notify observer
notify observer
notify observer
Observer A
Observer B
Observer C

***Figure 15.2**: Notifying observers*

The observers receive an update whenever there is a change in state. The subject notifies all the active observers by calling the update method of the observer interface.

Figure 15.3 exhibits the architecture of the observable design pattern. It describes the methods provided by the observable and the observer interfaces and their concrete implementations:

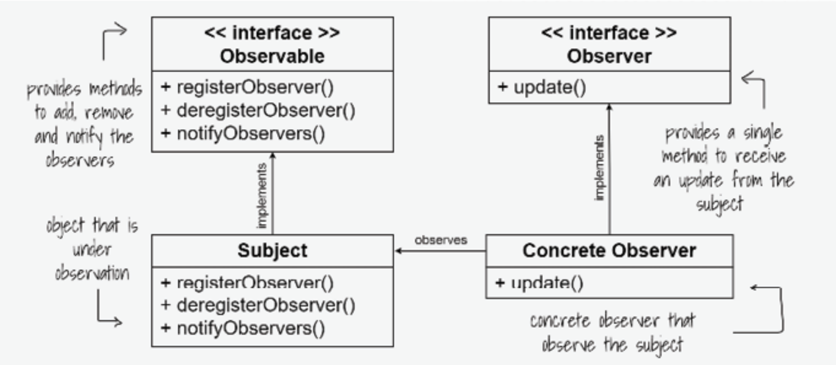

<< interface >>
Observable
+ registerObserver()
+ deregisterObserver()
+ notifyObservers()
<< interface >>
Observer
+ update()
provides methods to add, remove and notify the observers
implements
implements
provides a single method to receive an update from the subject
object that is under observation
Subject
+ registerObserver()
+ deregisterObserver()
+ notifyObservers()
observes
Concrete Observer
+ update()
concrete observer that observe the subject

***Figure 15.3**: The observer design pattern*

Here is how it works:

- Instantiate an object of type observer
- Register that object with the observable
- The observable notifies all the observers, including the one we registered in the step above
- When the observer receives an update, it can process that information as required

The observer design pattern can transfer information in two ways: push or pull. The information can be directly transferred to the observers whenever there is any change to the subject using the push mechanism. The pull mechanism, on the other hand, can be used to allow the observers to fetch the information in the form and shape they expect. The subject still notifies the observers about any change in the state. The push and pull mechanisms will be discussed in detail in the upcoming sections of the chapter.

The traffic light on a cross-section is a great example of the observer and the subject. The motorists and the riders all observe the change in the traffic light when they are at the cross-section. A change in the signal from red to green or green to yellow or from yellow to red is visible to all the motorists and the riders, which helps them decide whether they should wait at the cross-section or drive ahead.

Implementing observer pattern

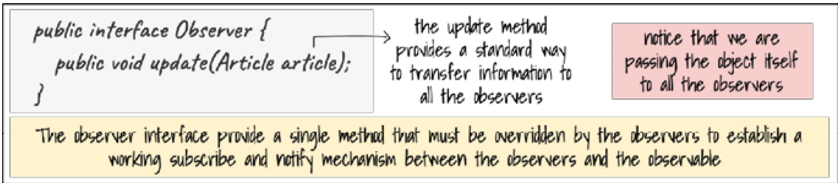
The example that we will walk you through in this chapter is of a virtual auction event. The auction event is supposed to host multiple buyers for the articles that will be up for sale during the event. An auctioneer will virtually oversee the proceedings to introduce the article to all the buyers and to sell it to the highest bidder. The publish and subscribe mechanism that we will develop through this example should take care of notifying the auctioneer and the registered buyers about the articles that are currently up for sale, one article at a time.

The example makes use of interfaces and their concrete implementations, which together form the publish and subscribe mechanism to support this event. The interfaces form the basis for setting up the observers and the observable. We will learn about implementing the observer design pattern and how it can be used to build a publish-subscribe model to broadcast changes in a controlled manner. Let us first look at the interfaces that form the backbone of the observer design pattern, and later we will discuss the concrete implementations of them to set up a working subscription mechanism for the virtual auction.

Interfaces

The observer design pattern provides two interfaces:

1. The **Observer** interface provides a single method that is used by the subject to update the observers whenever there is a change in the state of the subject, as shown in *Figure 15.4*:



```
public interface Observer {
    public void update(Article article);
}
```

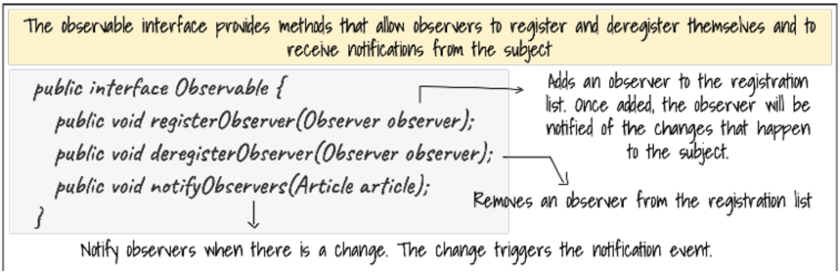
the update method provides a standard way to transfer information to all the observers

notice that we are passing the object itself to all the observers

The observer interface provide a single method that must be overridden by the observers to establish a working subscribe and notify mechanism between the observers and the observable

***Figure 15.4**: The observer interface*

1. The **Observable** interface provides methods that are used to register and deregister observers from the subject and to notify the observers about a change, as shown in *Figure 15.5*:



The observable interface provides methods that allow observers to register and deregister themselves and to receive notifications from the subject
Adds an observer to the registration list. Once added, the observer will be notified of the changes that happen to the subject.
Removes an observer from the registration list
Notify observers when there is a change. The change triggers the notification event.

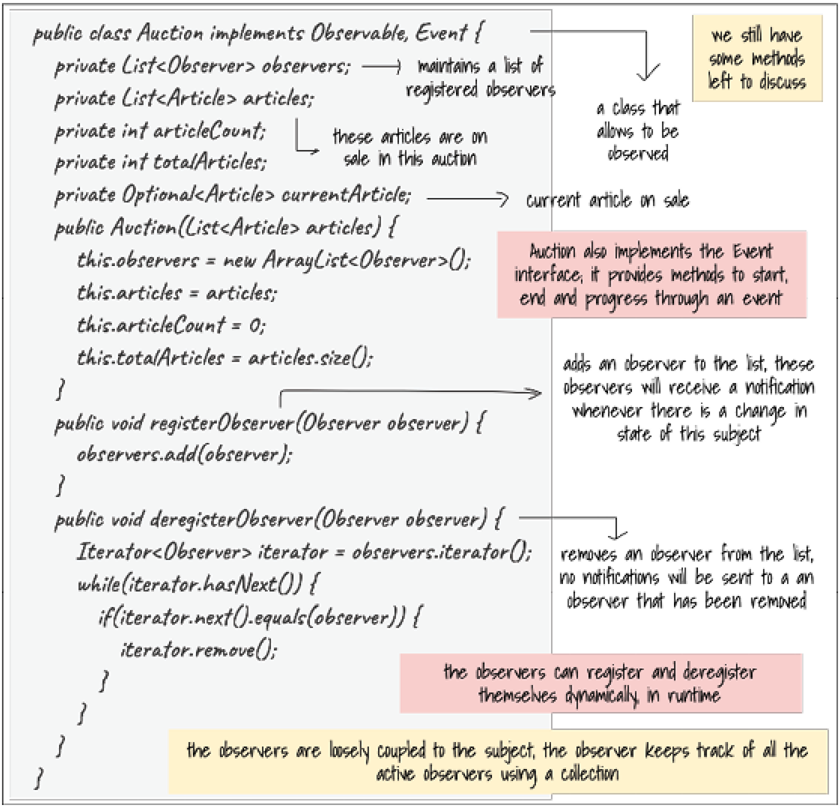
```
public interface Observable {
    public void registerObserver(Observer observer);
    public void deregisterObserver(Observer observer);
    public void notifyObservers(Article article);
}
```

***Figure 15.5**: The observer interface*

Subject

The auction must allow buyers to register and deregister, and it must notify the registered buyers about the change in the article that is currently being auctioned. To support all of this, our auction class implements the observable interface.

In *Figure 15.6*, we have showcased the implementation of our auction class. It maintains a list of all the active observers. Whenever an observer is registered with or deregistered from the auction, this list is updated. It also maintains a list of articles that will be up for sale during the event. The figure also exhibits a working definition of the register and deregister methods that are overridden by the auction implementation.



```
public class Auction implements Observable, Event {
   private List<Observer> observers;
   private List<Article> articles;
   private int articleCount;
   private int totalArticles;
   private Optional<Article> currentArticle;
   public Auction(List<Article> articles) {
      this.observers = new ArrayList<Observer>();
      this.articles = articles;
      this.articleCount = 0;
      this.totalArticles = articles.size();
   }
   public void registerObserver(Observer observer) {
      observers.add(observer);
   }
   public void deregisterObserver(Observer observer) {
      Iterator<Observer> iterator = observers.iterator();
      while(iterator.hasNext()) {
         if(iterator.next().equals(observer)) {
            iterator.remove();
         }
      }
   }
}
```

we still have some methods left to discuss

maintains a list of registered observers

a class that allows to be observed

these articles are on sale in this auction

current article on sale

Auction also implements the Event interface, it provides methods to start, end and progress through an event

adds an observer to the list, these observers will receive a notification whenever there is a change in state of this subject

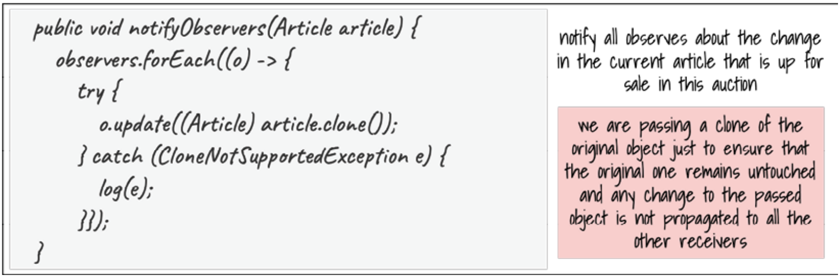
removes an observer from the list, no notifications will be sent to a an observer that has been removed

the observers can register and deregister themselves dynamically in runtime

the observers are loosely coupled to the subject, the observer keeps track of all the active observers using a collection

***Figure 15.6**: The Subject | Register and deregister*

Figure 15.7 exhibits a working definition of the **notify** method, the third method from the **Observable** interface. It traverses through every active observer and calls the update method on each of them:



```
public void notifyObservers(Article article) {
   observers.forEach((o) -> {
      try {
         o.update((Article) article.clone());
      } catch (CloneNotSupportedException e) {
         log(e);
      }});
}
```

notify all observes about the change in the current article that is up for sale in this auction

we are passing a clone of the original object just to ensure that the original one remains untouched and any change to the passed object is not propagated to all the other receivers

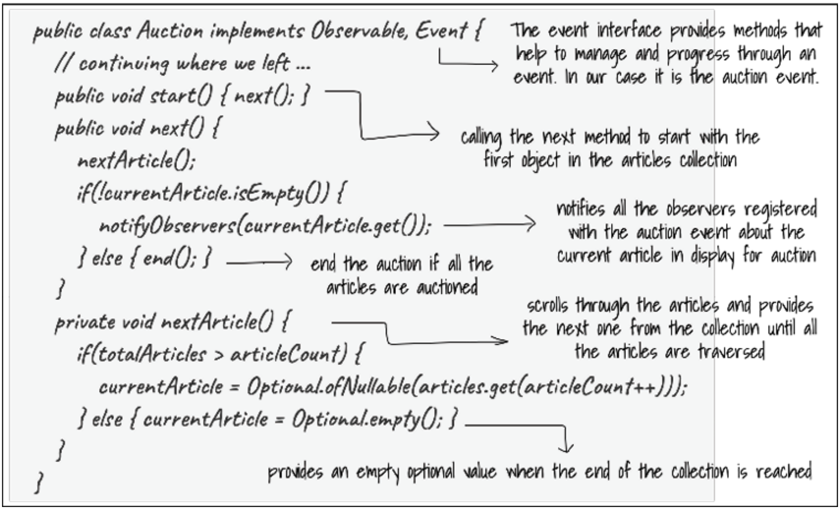
***Figure 15.7**: The Subject | Notify*

Notice that we have created a clone of the original **article** object.

This is to ensure that any change made by an observer to this object does not affect other observers.

Since the auction is also an event that must start and come to an end, it therefore inherits the event interface that allows for setting up and managing an event. The event interface is not related to the observer design pattern; it is used to demonstrate the working of notifications and the event that triggers them. The event interface provides the trigger points that allow for selecting the article that is next in line for auction and subsequently sending that information to the registered buyers. The buyers are updated about the next article through notifications that are sent by calling the update method on each buyer instance.

Figure 15.8 exhibits the definition of the methods inherited from the event interface. The **next** method does most of the work for us; it is this method that is called whenever the next article is to be made available for auction. It fetches the next available article from the list and notifies each buyer about that article by calling the notify method on all the buyer instances. The event is concluded if there are no more articles available to auction.



```
public class Auction implements Observable, Event {
   // continuing where we left ...
   public void start() { next(); }
   public void next() {
      nextArticle();
      if(!currentArticle.isEmpty()) {
         notifyObservers(currentArticle.get());
      } else { end(); }
   }
   private void nextArticle() {
      if(totalArticles > articleCount) {
         currentArticle = Optional.ofNullable(articles.get(articleCount++));
      } else { currentArticle = Optional.empty(); }
   }
}
```

The event interface provides methods that help to manage and progress through an event. In our case it is the auction event.

calling the next method to start with the first object in the articles collection

notifies all the observers registered with the auction event about the current article in display for auction

end the auction if all the articles are auctioned

scrolls through the articles and provides the next one from the collection until all the articles are traversed

provides an empty optional value when the end of the collection is reached

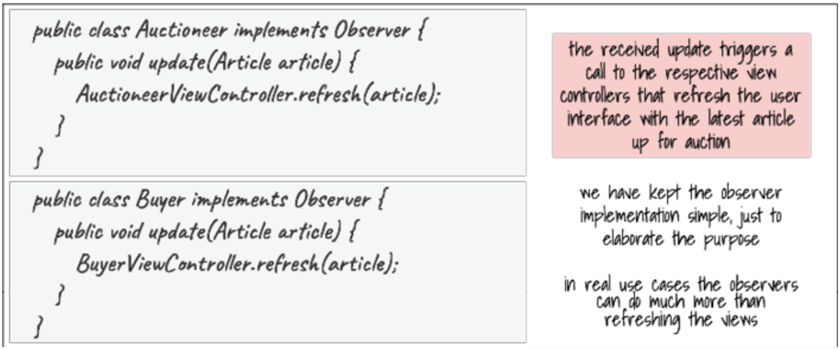
Figure 15.8: *The Subject | Event*

A call to the **next** method triggers the event that updates the next article up for auction, and that triggers the notifications to the registered buyers.

Observers

The buyers who registered for the auction expect to be notified about the articles when they are made available for sale. The buyers are observing the auction event. Apart from the buyers, we also have another observer, the auctioneer, who drives and supervises the virtual auction event.

Figure 15.9 exhibits the implementation of the **Observer** interface by the auctioneer and the buyer alike. The **update** method is called by the subject to send updates to the observers.



```
public class Auctioneer implements Observer {
    public void update(Article article) {
        AuctioneerViewController.refresh(article);
    }
}
```

the received update triggers a call to the respective view controllers that refresh the user interface with the latest article up for auction

```
public class Buyer implements Observer {
    public void update(Article article) {
        BuyerViewController.refresh(article);
    }
}
```

we have kept the observer implementation simple, just to elaborate the purpose

in real use cases the observers can do much more than refreshing the views

***Figure 15.9**: The observers*

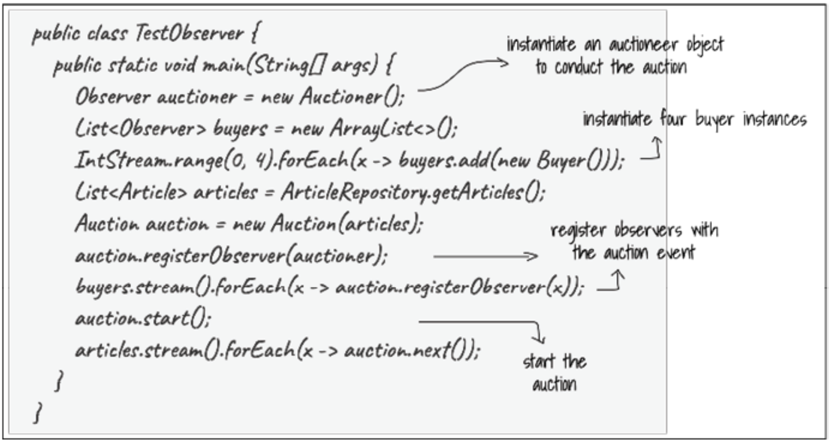
The implementations are kept simple to emphasize the design aspect. The observers are not just limited to receiving the data and sending it forward, as in *Figure 15.9*; instead, they can process it, store it, do complex calculations on it, and much more.

Testing the observer design pattern

We have written a test class to verify the working of the observer design pattern. Let us execute the same.

In *Figure 15.10*, we first instantiate our observers, that is, an instance of the auctioneer and a few buyer instances. Then we fetch the articles that will be up for sale from the article repository. Once we have the list of articles, we instantiate an auction object, passing to it the list of articles to set up the auction event. Then we register the auctioneer and all the buyers as observers to the auction event.

The auction event is started by calling the start method. To continue with the auction, we are traversing through all the available articles one by one, each time calling the **next** method on the auction. At last, when all the articles have been traversed, the auction will be concluded.



```
public class TestObserver {
   public static void main(String[] args) {
      Observer auctioner = new Auctioner();
      List<Observer> buyers = new ArrayList<>();
      IntStream.range(0, 4).forEach(x -> buyers.add(new Buyer()));
      List<Article> articles = ArticleRepository.getArticles();
      Auction auction = new Auction(articles);
      auction.registerObserver(auctioner);
      buyers.stream().forEach(x -> auction.registerObserver(x));
      auction.start();
      articles.stream().forEach(x -> auction.next());
   }
}
```

instantiate an auctioneer object to conduct the auction

instantiate four buyer instances

register observers with the auction event

start the auction

Figure 15.10: Test observer design pattern

The buyers will be updated about the new article from time to time whenever the next method on the auction is called.

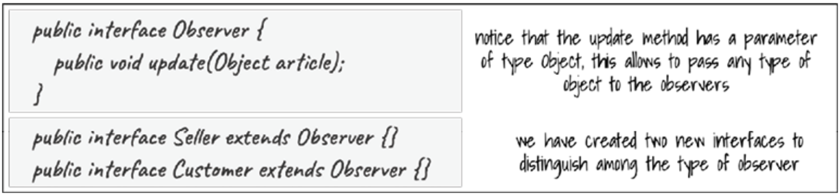
Varying implementations

The observer design pattern has two variations: the push variation and the pull variation. Both use the same interfaces and have similar implementations, apart from a small difference that separates them.

Push mechanism

This variation of the observer design pattern states that the subject should be responsible for sending the desired information to its observers. The subject should know the type and form of information required by its observers and should implement the logic to construct it in that form. It should be the responsibility of the subject to ensure that the observers receive the right information and do not have to implement any additional logic to convert it to the desired form.

Figure 15.11 describes the changes and additions required to support the push mechanism in the previous example:



```
public interface Observer {
    public void update(Object article);
}
```

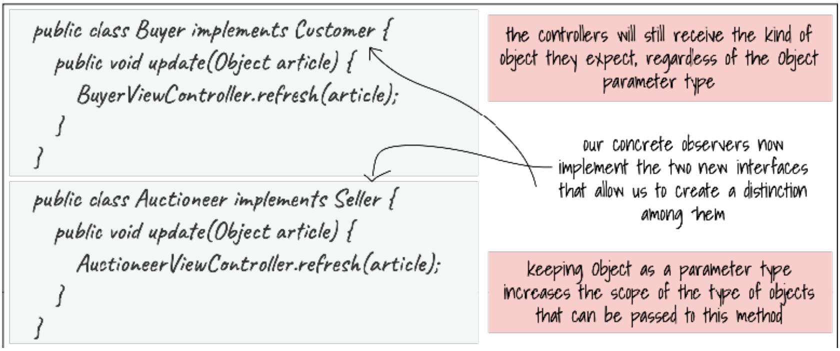
```
public interface Seller extends Observer {}
public interface Customer extends Observer {}
```

notice that the update method has a parameter of type Object, this allows to pass any type of object to the observers

we have created two new interfaces to distinguish among the type of observer

***Figure 15.11**: Push Mechanism | Observers I*

As shown in *Figure 15.12*, we have added two new interfaces to distinguish between the types of observers. The existing observer implementations now implement one of them each:



```
public class Buyer implements Customer {
    public void update(Object article) {
        BuyerViewController.refresh(article);
    }
}
```

```
public class Auctioneer implements Seller {
    public void update(Object article) {
        AuctioneerViewController.refresh(article);
    }
}
```

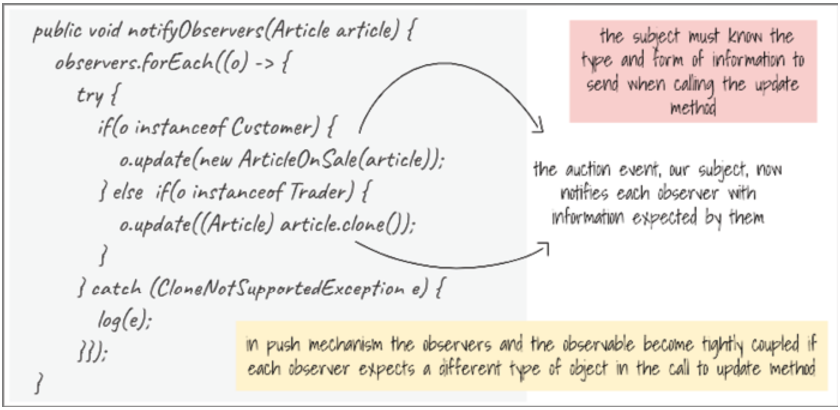
the controllers will still receive the kind of object they expect, regardless of the Object parameter type

our concrete observers now implement the two new interfaces that allow us to create a distinction among them

keeping Object as a parameter type increases the scope of the type of objects that can be passed to this method

***Figure 15.12**: Push mechanism | Observers II*

Figure 15.13 exhibits the updated definition of the notify method. The **notify** method now checks the type of observer before notifying them of the update, in the form expected by each of them:



```
public void notifyObservers(Article article) {
  observers.forEach((o) -> {
    try {
      if(o instanceof Customer) {
        o.update(new ArticleOnSale(article));
      } else  if(o instanceof Trader) {
        o.update((Article) article.clone());
      }
    } catch (CloneNotSupportedException e) {
      log(e);
    }});
}
```

the subject must know the type and form of information to send when calling the update method

the auction event, our subject, now notifies each observer with information expected by them

in push mechanism the observers and the observable become tightly coupled if each observer expects a different type of object in the call to update method

***Figure 15.13**: Push mechanism | Notification*

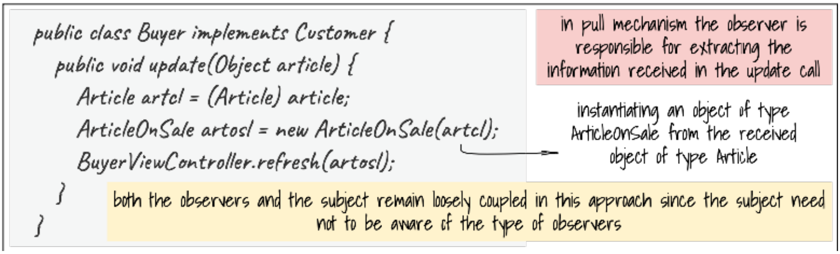
The requirement for the subject to know about the different types of observers and their expected form of information binds them in a tightly coupled relationship.

Pull mechanism

This variation of the observer design pattern states that the observers should pull the required information from what is passed to them via notification. The subject should pass on the entire information when they update the observers, and the observers should decide and fetch the relevant information as per their requirements.

This variation does not require changes in the subject but in the observers. The responsibility of processing the information received from the subject now rests with the observers. This keeps the relationship between the subject and observers loosely coupled, and the subject is no longer required to process the information for every different kind of observer. It does not even need to know the types of observers. All that the subject must ensure is the delivery of information whenever a change is triggered.

Figure 15.14 demonstrates an updated implementation of one of the observers observing the auction instance:



```
public class Buyer implements Customer {
   public void update(Object article) {
      Article artcl = (Article) article;
      ArticleOnSale artosl = new ArticleOnSale(artcl);
      BuyerViewController.refresh(artosl);
   }
}
```

in pull mechanism the observer is responsible for extracting the information received in the update call

instantiating an object of type ArticleOnSale from the received object of type Article

both the observers and the subject remain loosely coupled in this approach since the subject need not to be aware of the type of observers

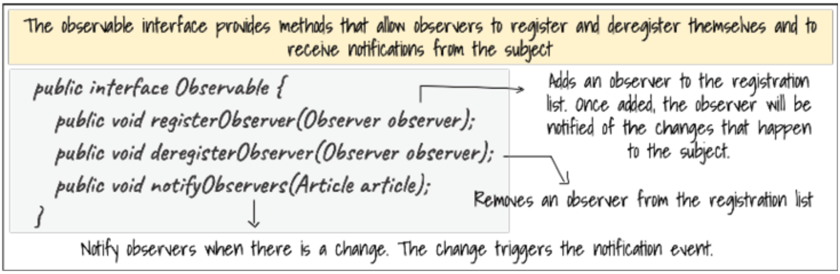
***Figure 15.14**: Pull mechanism | Observer*

It converts received information into the desired form before using it for further processing.

Summary

The observer design pattern allows for setting up a subscription model for information sharing where multiple observers observe a subject. The subject maintains a list of registered observers and notifies them of any change in state.

Figure 15.15 presents a high-level view of the observer design pattern using the constructs we used to demonstrate the same in this chapter. It presents a consolidated view of those constructs and the relation between them. It also mentions the necessary methods required for the pattern to function correctly:



The observable interface provides methods that allow observers to register and deregister themselves and to receive notifications from the subject

```
public interface Observable {
   public void registerObserver(Observer observer);
   public void deregisterObserver(Observer observer);
   public void notifyObservers(Article article);
}
```

Adds an observer to the registration list. Once added, the observer will be notified of the changes that happen to the subject.

Removes an observer from the registration list

Notify observers when there is a change. The change triggers the notification event.

***Figure 15.15**: Observer design pattern—defined*

Advantages and drawbacks

It is time to make note of some of the benefits and drawbacks of the observer design pattern. Let us start with the benefits first and then the drawbacks. Some of the benefits are as follows:

- The interacting objects, that is, the subject and the observers, are loosely coupled in a push mechanism.
- Observers can be added and removed at runtime.
- It broadcasts the information to multiple observers.

The drawbacks of the observer method design pattern are as follows:

- The observers are notified in a random order.
- The registration and deregistration process can cause memory leaks.

Usage

The following list outlines some of the scenarios that are relevant to the observer design pattern:

- When the list of observers is dynamic in nature, it can shrink and grow in runtime.
- When the information is to be broadcast to multiple receivers.

Conclusion

Observer design pattern is a widely used behavioral design pattern that can be utilized to prepare a subscription-based mechanism for transferring information. There are two ways to implement this pattern: push and pull. Push mechanism states that the subject should be responsible for transferring the correct form of information to the observers, whereas the pull mechanism states that the observers should be responsible for fetching the correct form of information from what is received in the notification.

In the next chapter, we will learn about the state design pattern and how it can be utilized to allow an object to alter its behavior when its state changes.

Questions

In this section, we will sharpen our brains by indulging in some fun brain activities. Let us get started.

Find the odd one out

1. Push, pull, tightly coupled, loosely coupled
2. Register, deregister, update, notify
3. Update, observer, observable, pull
4. Observer design pattern
    a. Broadcasts information
    b. Supports multiple subject
    c. Restricts notification to a single observer
    d. Cannot be used with multiple observers
5. Observer design pattern:
    a. Can be implemented in two ways
    b. Supports data transfer from observer to subject
    c. Holds on to the information if not queried for
    d. Is a structural design pattern
6. The notification from the subject:
    a. Reaches all the observers at the same time
    b. Is only sent to registered observers
    c. Reached the observers in a specific order
    d. Is only sent when the observer queries for it

Select two correct answers

1. Observer design pattern
    a. Push and pull mechanism
    b. Broadcasts information in a controlled manner
    c. Cannot be used with inheritance
    d. Reduces network latency
2. The observers
    a. Can register and deregister in runtime
    b. Can only connect to a single subject at a time

    c. Can choose to not to receive notification when subscribed
    d. Inherit a standard interface
3. What goes together?
    a. Push, pull, observer design pattern, broadcast
    b. Structural, observer design pattern, loose coupling
    c. Observer, observable, notification, subscription
    d. Push, observer, broadcast, structural

Answers

This section provides the answers to the questions given.

Find the odd one out

1. Pull
2. Update
3. Observable
4. Restricts notification to a single observer
5. Is a structural design pattern
6. Is only sent to registered observers

Select two correct answers

1. Push and pull mechanism, broadcasts information in a controlled manner
2. Can register and deregister in runtime, inherit a standard interface
3. Push, pull, observer design pattern, broadcast, and observer, observable, notification, subscription

Join our Discord space

Join our Discord workspace for latest updates, offers, tech happenings around the world, new releases, and sessions with the authors:

https://discord.bpbonline.com

CHAPTER 16

State and Behaviors

Introduction

At any moment, there is a finite number of states an object can be in. These states govern the behavior that the object exhibits when an operation is performed on it. This concept is close to that of a finite state machine. A finite state machine can only be in one of a finite number of states at any moment in time. It behaves differently in different states and can be made to switch between those states instantaneously. The transition from one state to another is predetermined, and hence, the machine may or may not switch to certain states depending on the current state of the machine.

As described in *Figure 16.1*, a state machine is often implemented using conditional operators to allow it to exhibit correct behavior and to switch between the states:

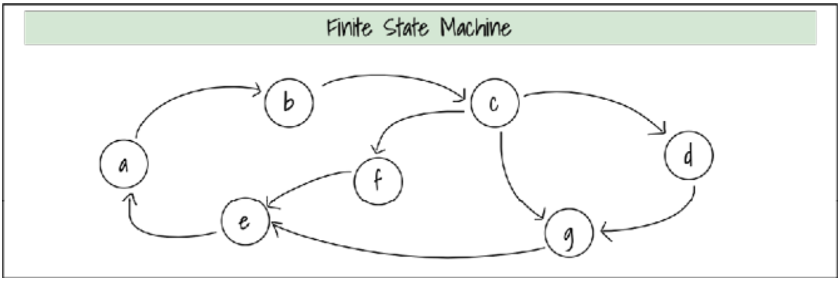
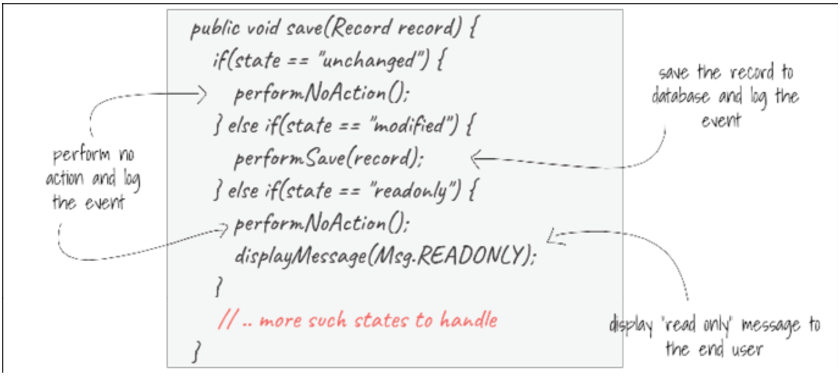


***Figure 16.1**: Finite state machine*

The class definition of a state machine uses multiple conditional statements to describe the flow of the algorithm. A limited set of conditions can be handled easily, but the addition of new states and dependent behaviors increases the complexity of the class definition. After a while, it becomes difficult to manage such an implementation, as any change to the existing transition logic may require changes to the existing methods that encapsulate the distinct behaviors of that object.

Frequent changes to the class definition keep it open for modification, an opportunity for bugs to creep in unnoticed.

Figure 16.2 exhibits a method that uses multiple conditional statements to perform correct operation based on the current state of the object:



```
public void save(Record record) {
  if(state == "unchanged") {
    performNoAction();
  } else if(state == "modified") {
    performSave(record);
  } else if(state == "readonly") {
    performNoAction();
    displayMessage(Msg.READONLY);
  }
  // .. more such states to handle
}
```

perform no action and log the event

save the record to database and log the event

display 'read only' message to the end user

***Figure 16.2**: Managing object states using conditional statements*

In this chapter, we will discuss the state design pattern and how it can be utilized to express states in the form of objects that encapsulate the distinct behaviors of the subject they are bound to.

We will learn about the power of delegation and how composition plays a big role in the state design pattern.

Structure

This chapter covers the following topics:

- The state design pattern
- Implementing the state pattern
- Summary
- Advantages and drawbacks
- Usage

Objectives

By the end of this chapter, the readers will be able to understand the complexity involved in maintaining the behaviors of an object under varying conditions and how to utilize the state design pattern to streamline it.

The chapter will discuss the implementation details of the state design pattern, the benefits it provides, and a few drawbacks, along with some of the places where this design pattern can be used.

The state design pattern

The state design pattern is a behavioral design pattern that allows an object to alter its behavior whenever its internal state changes. The state design pattern promotes the single responsibility principle. It advocates segregating the conditional behavior of an object into distinct classes, that is, creating classes for each state of that object and encapsulating the state-specific behavior in them. This approach provides a clean and simplified way to work with multiple states of an object and makes it easier to transition between them, subsequently changing the object's behavior. In short, it eliminates the need for conditional statements to decide the correct behavior of an object.

Figure 16.3 exhibits the architecture of the state design pattern:

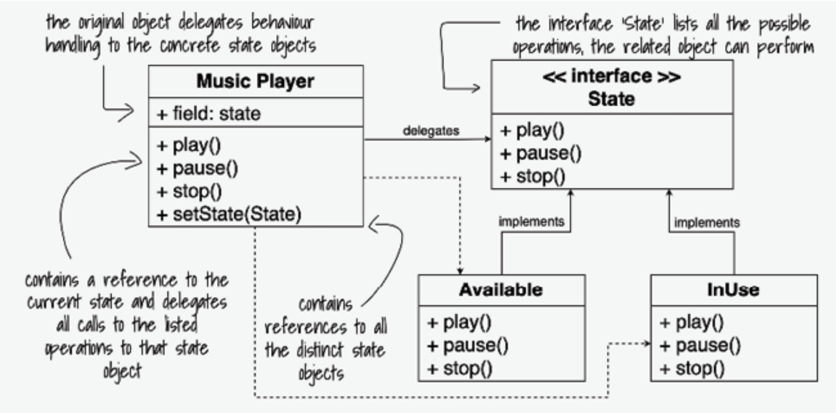

the original object delegates behaviour handling to the concrete state objects
the interface 'State' lists all the possible operations, the related object can perform
Music Player
+ field: state
+ play()
+ pause()
+ stop()
+ setState(State)
delegates
<< interface >>
State
+ play()
+ pause()
+ stop()
implements
implements
contains a reference to the current state and delegates all calls to the listed operations to that state object
contains references to all the distinct state objects
Available
+ play()
+ pause()
+ stop()
InUse
+ play()
+ pause()
+ stop()

Figure 16.3: *The state design pattern*

The interface state acts as a pivot and supports the conversion of a stateful object from a complex, condition-driven, single-class architecture to a streamlined, composition-based architecture. Each implementation of the interface is responsible for a specific state and the respective object behavior.

The stateful object maintains a reference to its current state and uses it to perform operations. Instead of implementing the behaviors by itself, the stateful object delegates calls to the listed operations to the current state. The current state is managed with a polymorphic reference that can point to multiple concrete implementations at different points in time.

Having a distinct implementation for each state of an object makes it easier to introduce new states, as it does not affect any existing behavior that is encapsulated in other state-specific implementations. The only required change is made to the definition of the stateful object itself. For the new state to be operational, it is added to the class definition of the stateful object. The state design pattern promotes the open-closed principle by encapsulating distinct behaviors in state-specific classes.

The boarding pass kiosk at airports is a great example of the state design pattern. The kiosk has a set of predetermined states that are observed by the user when the kiosk is used. These states govern the behavior of the kiosk when various operations are performed on it.

Implementing the state pattern

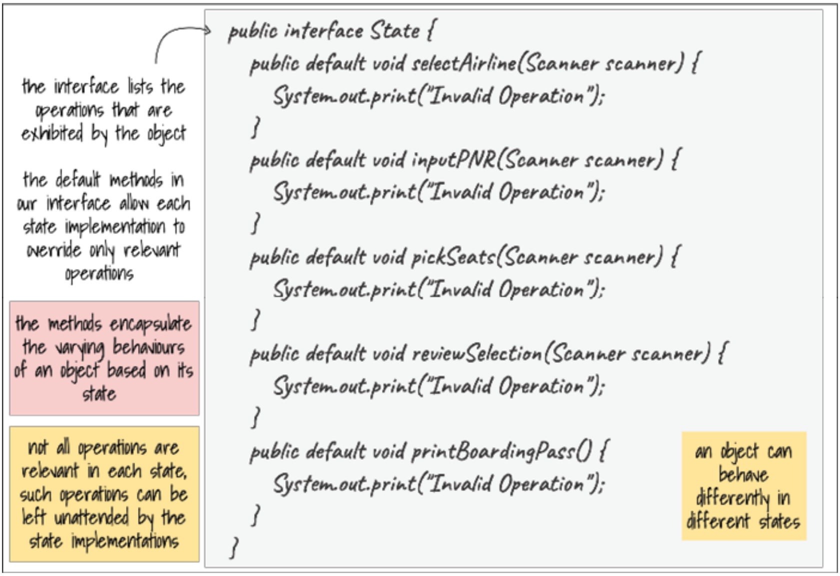
The example that we will walk you through in this chapter is an imitation of the boarding pass kiosk that we discussed in the preceding section. Our example makes use of an interface and its concrete implementations that form the distinct states of the kiosk. The concrete implementations need not override operations that are not relevant to them, as the interface provides the default definition for those.

Let us first look at the interface that forms the backbone of the state design pattern, and in the following section, we will discuss the concrete implementations that encapsulate the behavior of the kiosk object.

The interface

The state design pattern offers a single interface called **state**. It provides exact replicas of the operations as the stateful object, that is, they have the same signatures. These methods are overridden by the concrete implementations to provide state-specific behaviors.

Figure 16.4 exhibits the state interface that we have created for our example. It contains five different operations that can be performed on the kiosk object. The interface provides a default implementation for each of the methods to ensure that the concrete implementations need not override operations that are invalid for the object state they represent. All the other operations are deemed valid for that state:



```
public interface State {
   public default void selectAirline(Scanner scanner) {
      System.out.print("Invalid Operation");
   }
   public default void inputPNR(Scanner scanner) {
      System.out.print("Invalid Operation");
   }
   public default void pickSeats(Scanner scanner) {
      System.out.print("Invalid Operation");
   }
   public default void reviewSelection(Scanner scanner) {
      System.out.print("Invalid Operation");
   }
   public default void printBoardingPass() {
      System.out.print("Invalid Operation");
   }
}
```

the interface lists the operations that are exhibited by the object

the default methods in our interface allow each state implementation to override only relevant operations

the methods encapsulate the varying behaviours of an object based on its state

not all operations are relevant in each state, such operations can be left unattended by the state implementations

an object can behave differently in different states

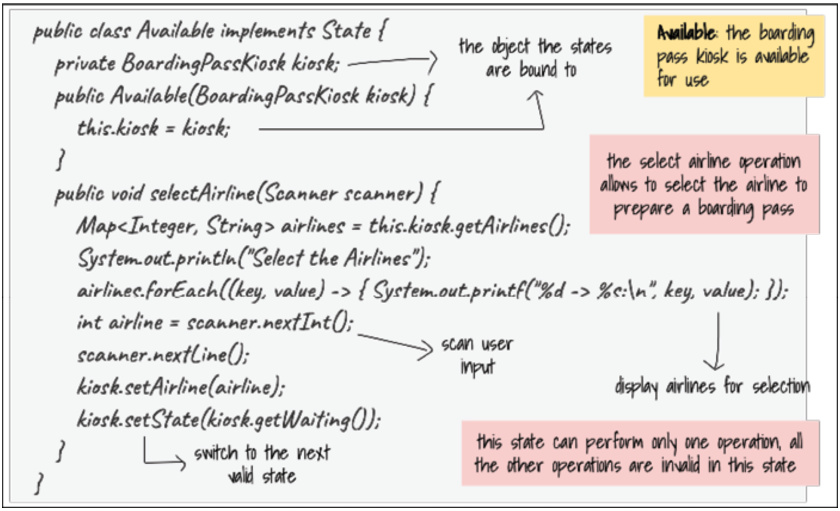
***Figure 16.4**: The state interface*

The concrete implementations

The concrete implementations of the state interface encapsulate the behavior of the stateful object in accordance with the object state that they express. Each of these implementations has a reference to the stateful object that is used to change the current state of that object.

Our example, which demonstrates the use of the state design pattern, uses five different states to illustrate the working of a boarding pass kiosk. These five states are called **available**, **waiting**, **verified**, **review**, and **confirm**. All of them implement the state interface and override the operations that exhibit a state-specific behavior.

Figure 16.5 exhibits the implementation of the initial state of the kiosk, that is, **Available**:



```java
public class Available implements State {
   private BoardingPassKiosk kiosk;
   public Available(BoardingPassKiosk kiosk) {
      this.kiosk = kiosk;
   }
   public void selectAirline(Scanner scanner) {
      Map<Integer, String> airlines = this.kiosk.getAirlines();
      System.out.println("Select the Airlines");
      airlines.forEach((key, value) -> { System.out.printf("%d -> %s:\n", key, value); });
      int airline = scanner.nextInt();
      scanner.nextLine();
      kiosk.setAirline(airline);
      kiosk.setState(kiosk.getWaiting());
   }
}
```

the object the states are bound to
Available: the boarding pass kiosk is available for use
the select airline operation allows to select the airline to prepare a boarding pass
scan user input
display airlines for selection
switch to the next valid state
this state can perform only one operation, all the other operations are invalid in this state

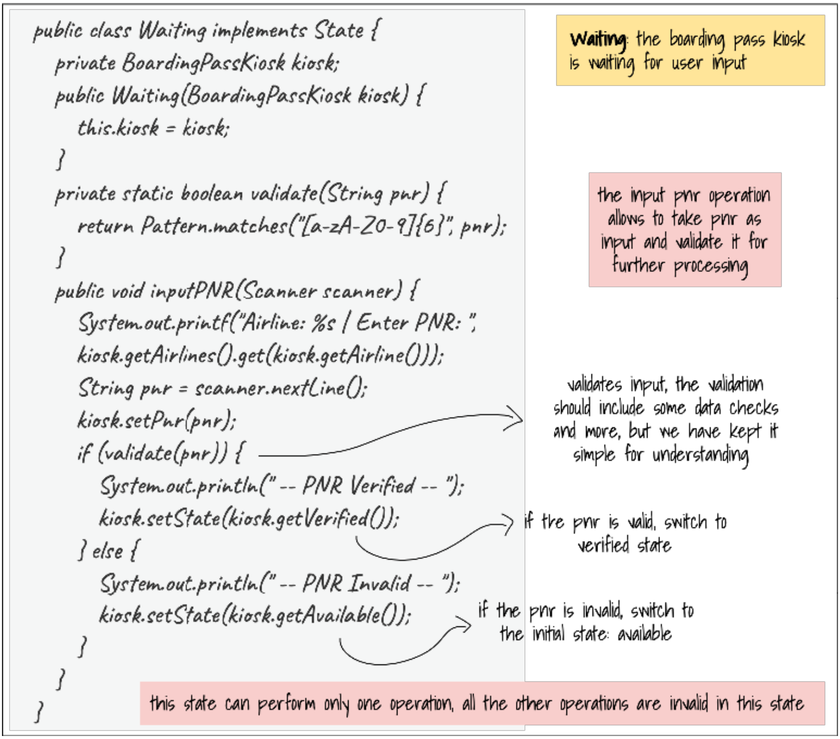
***Figure 16.5**: State—Available*

This state indicates that the kiosk is available for use to generate and print the boarding pass. The only valid operation in this state allows the selection of the airline to prepare a boarding pass. It presents the user with multiple options to select from, scans the user selection, and then switches the current state of the kiosk object to **Waiting**.

All the other operations are invalid and are not defined in this state implementation. If those operations were called from this state of the kiosk object, they would return with an invalid operation message. This holds true for all the other state implementations as well, for the operations that are not overridden by them.

After successful selection of the airline in the previous state, the kiosk switches to the **Waiting** state. This state indicates that the kiosk is waiting for the user to enter the PNR information of the airline ticket to prepare the boarding pass.

This state has only one valid operation. This operation takes the PNR as input, verifies it for correctness, and then makes a state transition based on the result of the verification process. If the PNR is validated to be correct, the current state of the kiosk is switched to **Verified**. If the verification process fails because of an incorrect PNR, the kiosk is switched back to the initial state, that is, **Available**, and then the user must restart the process. *Figure 16.6* exhibits the implementation of the next state of the kiosk, that is, **Waiting**:



```
public class Waiting implements State {
   private BoardingPassKiosk kiosk;
   public Waiting(BoardingPassKiosk kiosk) {
      this.kiosk = kiosk;
   }
   private static boolean validate(String pnr) {
      return Pattern.matches("[a-zA-Z0-9]{6}", pnr);
   }
   public void inputPNR(Scanner scanner) {
      System.out.printf("Airline: %s / Enter PNR: ",
      kiosk.getAirlines().get(kiosk.getAirline()));
      String pnr = scanner.nextLine();
      kiosk.setPnr(pnr);
      if (validate(pnr)) {
         System.out.println(" -- PNR Verified -- ");
         kiosk.setState(kiosk.getVerified());
      } else {
         System.out.println(" -- PNR Invalid -- ");
         kiosk.setState(kiosk.getAvailable());
      }
   }
}
```

Waiting: the boarding pass kiosk is waiting for user input

the input pnr operation allows to take pnr as input and validate it for further processing

validates input, the validation should include some data checks and more, but we have kept it simple for understanding

if tre pnr is valid, switch to verified state

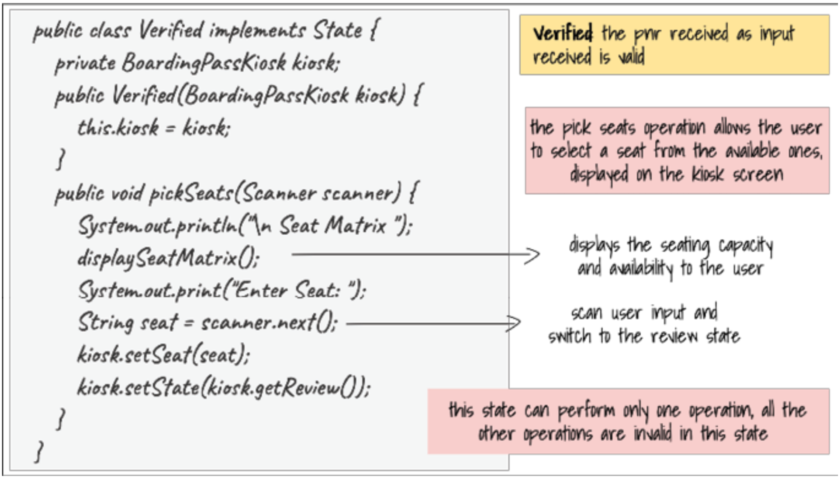
if the pnr is invalid, switch to the initial state: available

this state can perform only one operation, all the other operations are invalid in this state

***Figure 16.6**: State—Waiting*

To keep the example simple, the validation process only checks for the correctness of the entered PNR information using a preset regular expression. This is possibly the weakest form of verification. For a production-ready code, it must be ensured that the entered PNR not only follows the format but also is a valid PNR of the selected airline. Plenty of other validations can be made depending on the implementation.

After successful verification of the PNR in the previous state, the kiosk switches to the **Verified** state. This state indicates that the PNR has been verified and the kiosk is now waiting for the user to select a seat for her travel. This state, too, has only one valid operation. This operation displays the seat matrix to the user and takes her input to reserve that seat. To keep the example simple, we are assuming that the user will not select an already occupied seat. We are also not validating the selected seat's availability, though this must be ensured in a production-ready application. *Figure 16.7* exhibits the implementation of the **Verified** state:



```
public class Verified implements State {
   private BoardingPassKiosk kiosk;
   public Verified(BoardingPassKiosk kiosk) {
      this.kiosk = kiosk;
   }
   public void pickSeats(Scanner scanner) {
      System.out.println("\n Seat Matrix ");
      displaySeatMatrix();
      System.out.print("Enter Seat: ");
      String seat = scanner.next();
      kiosk.setSeat(seat);
      kiosk.setState(kiosk.getReview());
   }
}
```

Verified the pnr received as input received is valid

the pick seats operation allows the user to select a seat from the available ones, displayed on the kiosk screen

displays the seating capacity and availability to the user

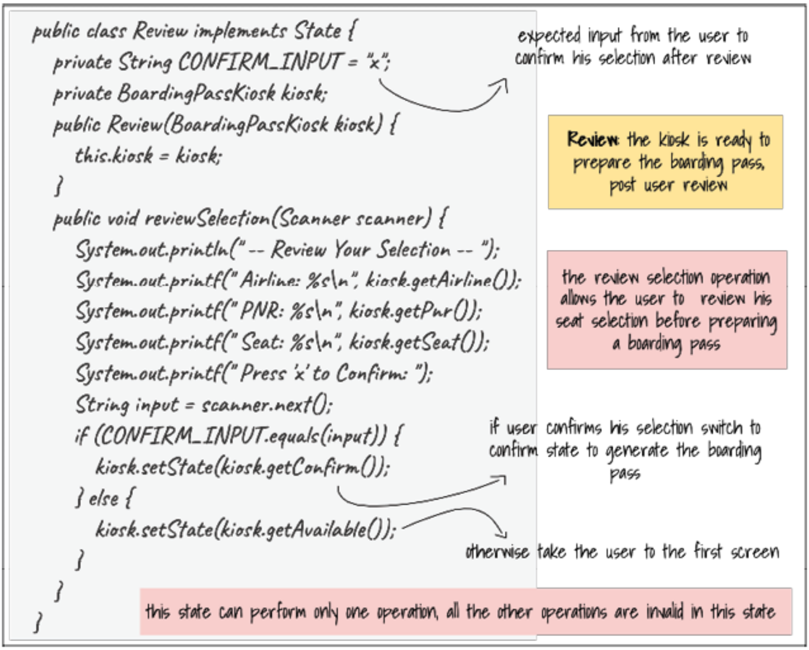
scan user input and switch to the review state

this state can perform only one operation, all the other operations are invalid in this state

***Figure 16.7**: State—Verified*

After successful selection of a seat in the previous state, the kiosk switches to the **Confirm** state. This state indicates that the kiosk is ready to prepare the boarding pass for the user and is waiting for her to review the selections made in the previous states.

This state, too, has only one valid operation. This operation displays the prepared information for the boarding pass to the user and takes her confirmation to finally print that boarding pass. *Figure 16.8* demonstrates the implementation of the **Review** state:



```
public class Review implements State {
   private String CONFIRM_INPUT = "x";
   private BoardingPassKiosk kiosk;
   public Review(BoardingPassKiosk kiosk) {
      this.kiosk = kiosk;
   }
   public void reviewSelection(Scanner scanner) {
      System.out.println("-- Review Your Selection -- ");
      System.out.printf("Airline: %s\n", kiosk.getAirline());
      System.out.printf("PNR: %s\n", kiosk.getPnr());
      System.out.printf("Seat: %s\n", kiosk.getSeat());
      System.out.printf("Press 'x' to Confirm: ");
      String input = scanner.next();
      if (CONFIRM_INPUT.equals(input)) {
         kiosk.setState(kiosk.getConfirm());
      } else {
         kiosk.setState(kiosk.getAvailable());
      }
   }
}
```

expected input from the user to confirm his selection after review

Review: the kiosk is ready to prepare the boarding pass, post user review

the review selection operation allows the user to review his seat selection before preparing a boarding pass

if user confirms his selection switch to confirm state to generate the boarding pass

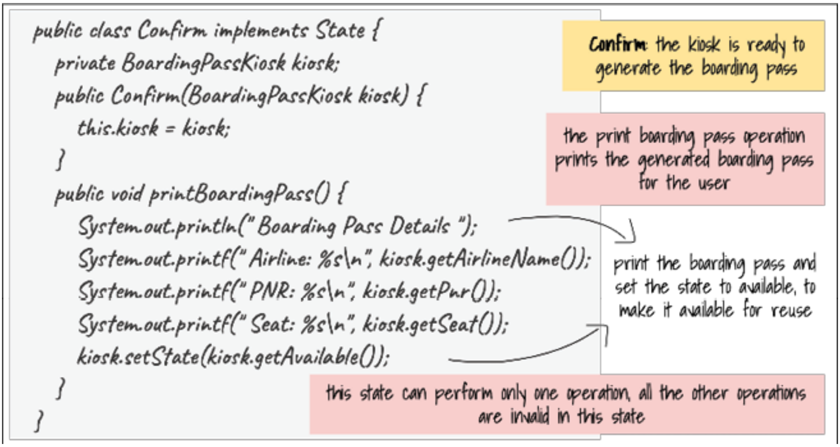
otherwise take the user to the first screen

this state can perform only one operation, all the other operations are invalid in this state

***Figure 16.8**: State—Review*

If the user confirms that the details of the boarding pass are correct, the current state of the kiosk is switched to **Confirm**; otherwise, the kiosk is switched back to the initial state, that is, **Available**.

Figure 16.9 demonstrates the final state of the boarding pass kiosk, that is, **Confirm**:



```
public class Confirm implements State {
   private BoardingPassKiosk kiosk;
   public Confirm(BoardingPassKiosk kiosk) {
      this.kiosk = kiosk;
   }
   public void printBoardingPass() {
      System.out.println("Boarding Pass Details ");
      System.out.printf("Airline: %s\n", kiosk.getAirlineName());
      System.out.printf("PNR: %s\n", kiosk.getPnr());
      System.out.printf("Seat: %s\n", kiosk.getSeat());
      kiosk.setState(kiosk.getAvailable());
   }
}
```

Confirm: the kiosk is ready to generate the boarding pass

the print boarding pass operation prints the generated boarding pass for the user

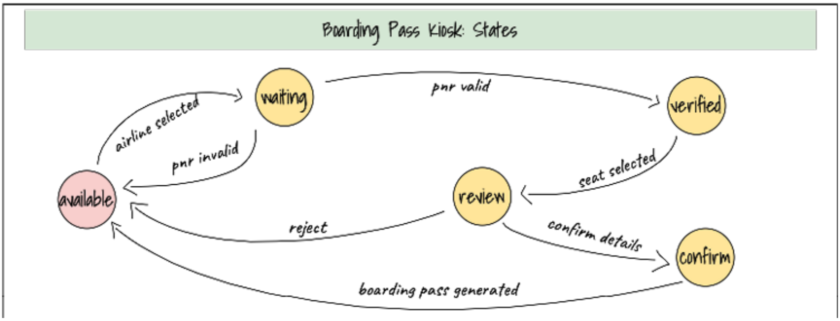
print the boarding pass and set the state to available, to make it available for reuse

this state can perform only one operation, all the other operations are invalid in this state

***Figure 16.9**: State—Confirm*

This state is an auto-transition state and does not require user intervention. This state provides only one valid operation that prints the prepared boarding pass and switches the current state of the kiosk to **Available** so that it can be used by another user to generate and print a boarding pass.

Figure 16.10 demonstrates the state diagram for the boarding pass kiosk:



Boarding Pass Kiosk: States
waiting
pnr valid
verified
airline selected
pnr invalid
seat selected
available
review
reject
confirm details
confirm
boarding pass generated

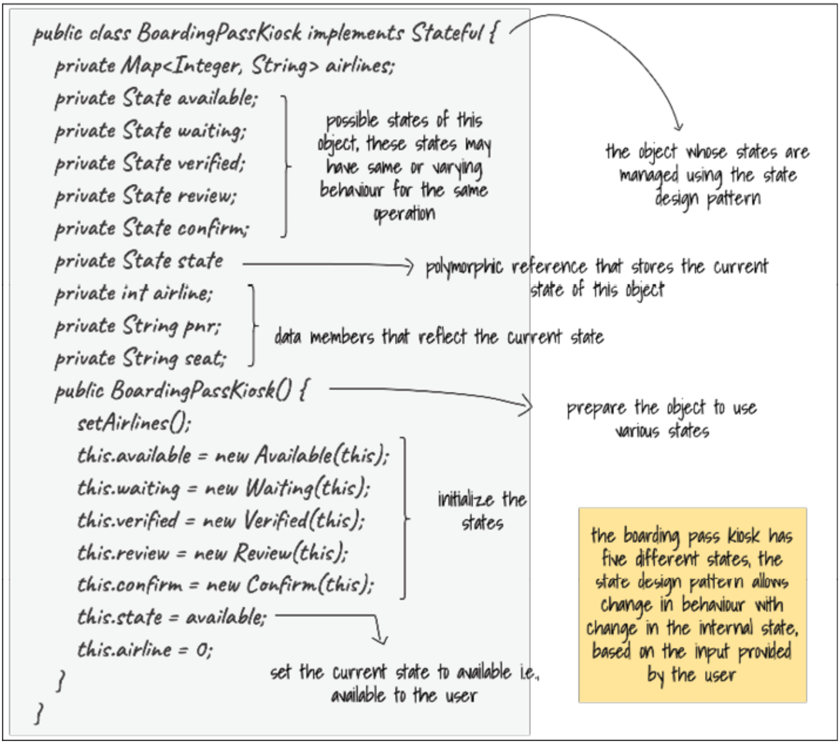
***Figure 16.10**: State diagram*

The stateful object

A stateful object could be any object that has different states and that exhibits distinct, state-specific behaviors for the operations that can be performed on it. It keeps a reference to its current state and delegates all the incoming requests to it.

It stores the value of its current state and has a few data members that reflect it. It also keeps references to all the possible states of itself and provides getter methods to access them. They are used within the concrete state implementations to switch to the next appropriate state. Keeping these references in the kiosk object makes it easier for the state implementations to drive the transition while being unaware of other states' existence. This also keeps these implementations loosely coupled.

The constructor is responsible for instantiating all the states and passes a reference of itself to them, to bind the state instances and the kiosk object together. There is no need to initialize any of the object’s state explicitly; it has already been taken care of by the constructor definition. *Figure 16.11* exhibits the data members and the constructor of the boarding pass kiosk:



```
public class BoardingPassKiosk implements Stateful {
   private Map<Integer, String> airlines;
   private State available;
   private State waiting;
   private State verified;
   private State review;
   private State confirm;
   private State state
   private int airline;
   private String pnr;
   private String seat;
   public BoardingPassKiosk() {
      setAirlines();
      this.available = new Available(this);
      this.waiting = new Waiting(this);
      this.verified = new Verified(this);
      this.review = new Review(this);
      this.confirm = new Confirm(this);
      this.state = available;
      this.airline = 0;
   }
}
```

the object whose states are managed using the state design pattern

possible states of this object, these states may have same or varying behaviour for the same operation

polymorphic reference that stores the current state of this object

data members that reflect the current state

prepare the object to use various states

initialize the states

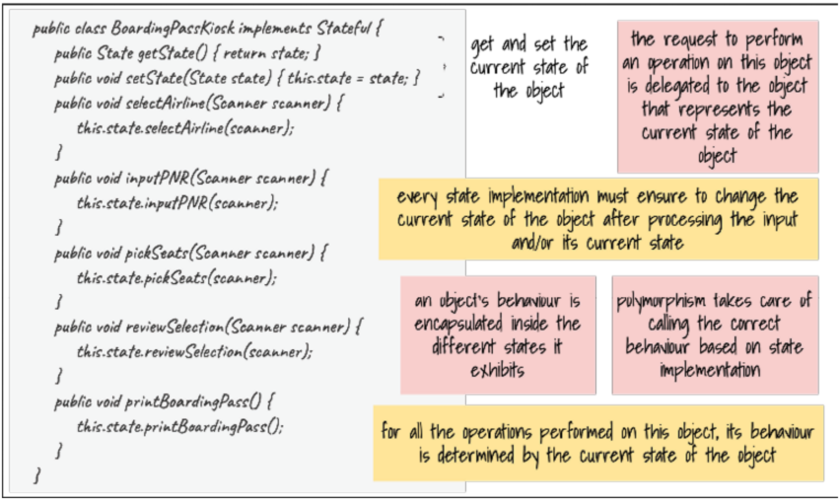
set the current state to available i.e., available to the user

the boarding pass kiosk has five different states, the state design pattern allows change in behaviour with change in the internal state, based on the input provided by the user

Figure 16.11: *The stateful object I*

The stateful object stores the value of its current state and has a few data members that reflect it. Notice that the operation definitions delegate the responsibility of performing that operation on the current state. This is much more readable and cleaner than having multiple conditional statements decide the current behavior of this object.

Figure 16.12 demonstrates the operations that can be performed on the kiosk object:



```
public class BoardingPassKiosk implements Stateful {
    public State getState() { return state; }
    public void setState(State state) { this.state = state; }
    public void selectAirline(Scanner scanner) {
        this.state.selectAirline(scanner);
    }
    public void inputPNR(Scanner scanner) {
        this.state.inputPNR(scanner);
    }
    public void pickSeats(Scanner scanner) {
        this.state.pickSeats(scanner);
    }
    public void reviewSelection(Scanner scanner) {
        this.state.reviewSelection(scanner);
    }
    public void printBoardingPass() {
        this.state.printBoardingPass();
    }
}
```

get and set the current state of the object

the request to perform an operation on this object is delegated to the object that represents the current state of the object

every state implementation must ensure to change the current state of the object after processing the input and/or its current state

an object's behaviour is encapsulated inside the different states it exhibits

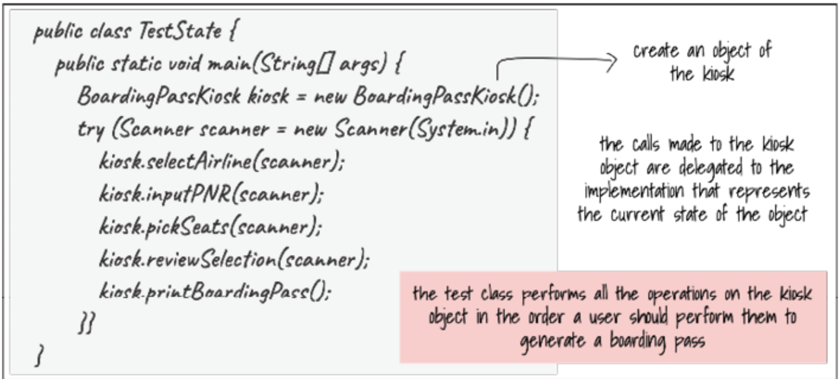
polymorphism takes care of calling the correct behaviour based on state implementation

for all the operations performed on this object, its behaviour is determined by the current state of the object

Figure 16.12: *The stateful object II*

Testing the state design pattern

We have written a test class to verify the working of the state design pattern. In *Figure 16.13*, we first instantiate our stateful object, that is, an instance of a boarding pass kiosk and later call different operations on it in the order that with the help of correct input values, guarantees the generation of a boarding pass kiosk.



```
public class TestState {
    public static void main(String[] args) {
        BoardingPassKiosk kiosk = new BoardingPassKiosk();
        try (Scanner scanner = new Scanner(System.in)) {
            kiosk.selectAirline(scanner);
            kiosk.inputPNR(scanner);
            kiosk.pickSeats(scanner);
            kiosk.reviewSelection(scanner);
            kiosk.printBoardingPass();
        }}
}
```

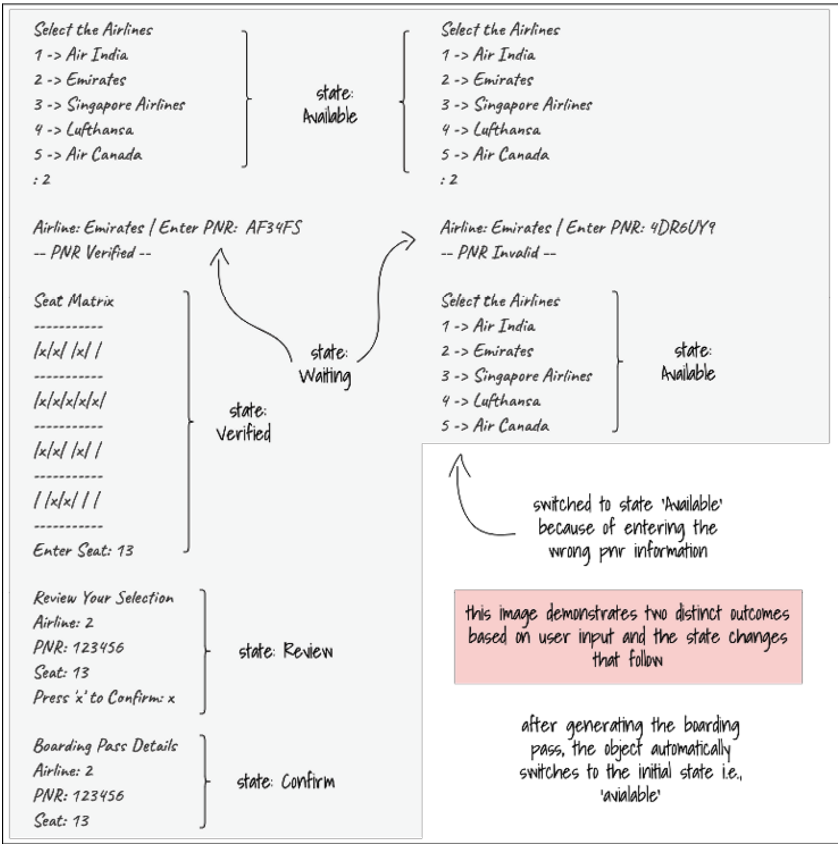
create an object of the kiosk

the calls made to the kiosk object are delegated to the implementation that represents the current state of the object

the test class performs all the operations on the kiosk object in the order a user should perform them to generate a boarding pass

Figure 16.13: *Test state design pattern*

Figure 16.14 exhibits two different outcomes based on user inputs and the state changes that follow. The left section of the figure demonstrates that by following the correct steps and entering the correct input, the user was able to generate a boarding pass from the kiosk. The right section demonstrates the outcome of entering wrong PNR information; the object's state switched back to **Available**:



```
Select the Airlines
1 -> Air India
2 -> Emirates
3 -> Singapore Airlines
4 -> Lufthansa
5 -> Air Canada
: 2

Airline: Emirates / Enter PNR: AF34FS
-- PNR Verified --

Seat Matrix
-----------
/x/x/ /x/ /
-----------
/x/x/x/x/x/
-----------
/x/x/ /x/ /
-----------
/ /x/x/ / /
-----------
Enter Seat: 13

Review Your Selection
Airline: 2
PNR: 123456
Seat: 13
Press 'x' to Confirm: x

Boarding Pass Details
Airline: 2
PNR: 123456
Seat: 13
```

```
Select the Airlines
1 -> Air India
2 -> Emirates
3 -> Singapore Airlines
4 -> Lufthansa
5 -> Air Canada
: 2

Airline: Emirates / Enter PNR: 4DR6UY9
-- PNR Invalid --

Select the Airlines
1 -> Air India
2 -> Emirates
3 -> Singapore Airlines
4 -> Lufthansa
5 -> Air Canada
```

switched to state 'Available' because of entering the wrong pnr information

this image demonstrates two distinct outcomes based on user input and the state changes that follow

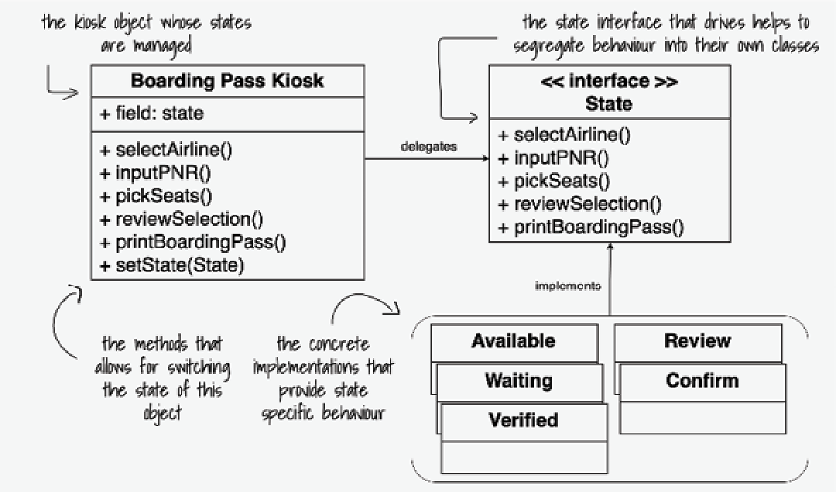
after generating the boarding pass, the object automatically switches to the initial state i.e., 'avialable'

Figure 16.14: Test state design pattern: Outcome

Summary

The state design pattern allows for managing the state-specific behaviors of an object by separating the behavior from the object definition. It promotes the principle of separation of concern and the single responsibility principle. It suggests defining state-specific behaviors in their own classes while keeping those definitions loosely coupled to each other. The state design pattern can be used to manage multiple states and switch between them effortlessly, at runtime.

Figure 16.15 exhibits a high-level view of the state design pattern:



the kiosk object whose states are managed
Boarding Pass Kiosk
+ field: state
+ selectAirline()
+ inputPNR()
+ pickSeats()
+ reviewSelection()
+ printBoardingPass()
+ setState(State)
delegates
the state interface that drives helps to segregate behaviour into their own classes
<< interface >>
State
+ selectAirline()
+ inputPNR()
+ pickSeats()
+ reviewSelection()
+ printBoardingPass()
implements
the methods that allows for switching the state of this object
the concrete implementations that provide state specific behaviour
Available
Waiting
Verified
Review
Confirm

***Figure 16.15**: State design pattern—Defined*

It presents a consolidated view of the relationship between the stateful object and the concrete state implementations.

Advantages and drawbacks

It is time to make note of some of the benefits and drawbacks of the state design pattern. Let us start with the benefits first:

- Makes it easier to introduce new states that support additional behavior.
- Minimizes the complexity of the stateful object by separating the behavior from the object definition and reducing the use of condition statements.
- Uses composition and promotes open/close principle that reduces the need for regression testing.

The drawbacks of the state method design pattern are as follows:

- A lot of code must be written and managed for the state design pattern
- An increased number of states can increase the memory

footprint if the state objects are initialized in abundance. Not something that the current systems need to be worried about.

Usage

The following list outlines some of the scenarios relevant to the state design pattern:

- When an object's behavior is dependent on its internal state, the state design pattern can help to segregate the behavior into specific classes.
- When an object's definition is polluted with conditional statements, the state design pattern can help to reduce the clutter and ease code maintenance.

Conclusion

State design pattern is an outstanding mechanism to reduce avoidable conditional checks and move the state-specific behavior into its own class. The behavior of the object can easily be changed by switching states, and new behaviors can be introduced without affecting existing state or behavior.

In the next chapter, we will learn about the command design pattern and how it can be utilized to encapsulate information needed to perform an action at a later stage.

Questions

In this section, we will sharpen our brains by indulging in some fun brain activities. Let us get started.

Find the odd one out

1. Composition, extension, loosely coupled, polymorphic
2. Single responsibility, separation of concern, interface segregation, open/close
3. Multiple states, conditional execution, distinct behavior, transition

4. State design pattern
    a. Allows managing state-specific behavior.
    b. Supports multiple states.
    c. Requires states to be directly aware of each other.
    d. Provides for switching object state at runtime.
5. State design pattern
    a. Depends on inheritance and polymorphism.
    b. Works on the principle of interface segregation.
    c. Segregates object behavior into state-specific classes.
    d. Is a behavioral design pattern
6. State transition using the state design pattern
    a. It can only be done using the base state implementation.
    b. It is a simple and clean approach.
    c. It can only be done in runtime.
    d. Requires the stateful object to expose a switch mechanism.

Select two correct answers

1. State design pattern:
    a. Cannot work without polymorphism
    b. Increases object complexity
    c. Promotes separation of concern
    d. Increases cohesion
2. The stateful object:
    a. Allows for switching states in runtime
    b. Encapsulate its behavior
    c. Delegates responsibility to current state
    d. Inherits a standard interface
3. What goes together?
    a. Encapsulation, concrete states, object behavior
    b. Structural, state design pattern, loose coupling

c. State design pattern, behavioral, loose coupling

d. Inheritance, extension, single state implementation

Answers

This section provides the answers to the questions given.

Find the odd one out

1. Extension
2. Interface segregation
3. Conditional execution
4. Requires states to be directly aware of each other
5. Works on the principle of interface segregation
6. Requires the stateful object to expose a switch mechanism

Select two correct answers

1. Cannot work without polymorphism, promotes the separation of concerns.
2. Delegates responsibility to the current state, inherits a standard interface.
3. Encapsulation, concrete states, object behavior, and state design pattern, behavioral, loose coupling.

CHAPTER 17

Executing Commands

Introduction

Software applications are increasingly providing a host of new options to their users. While it is good to have new features in the application that add to the product catalog, their implementation is not always straightforward. Imagine a user interface that allows the user to select from multiple options. The user interface is most likely to have buttons or menu items to provide access to these features, where each feature has a button dedicated to it.

One way to support all these features is to have multiple implementations, each dedicated to performing one single task, and a new button on the user interface for each of these features. Moreover, every **application programming interface** (**API**) that is accessed through these buttons may have a different interface requiring specific integrations. When the buttons should not really be aware of what they do, they remain coupled to their directives.

The interface and program logic are two different aspects of an application. An interface should not be tightly coupled to the functionality it serves; otherwise, it would be difficult to reuse any of it. Imagine duplicating the button code every time there is a new feature requirement. The code will become highly redundant, and the benefit of reusability will be lost.

Figure 17.1 exhibits the relationship between a button and the various services that can be triggered because of an event performed on that button:

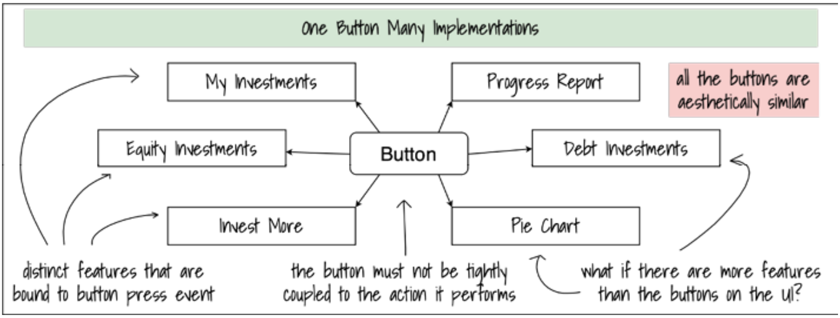

One Button Many Implementations
My Investments
Progress Report
all the buttons are aesthetically similar
Equity Investments
Button
Debt Investments
Invest More
Pie Chart
distinct features that are bound to button press event
the button must not be tightly coupled to the action it performs
what if there are more features than the buttons on the ui?

***Figure 17.1**: Multiple feature implementations*

It demonstrates that while all the buttons are aesthetically the same, binding them to specific tasks will take away the advantage of reusing them.

What we need is a way to decouple the button from the action it performs. The button does not need to know about the action that is performed and who performs it. It should only be responsible for providing a way to access some functionality whenever it is pressed, or some other event is performed on it. There should be a total separation of concern between these two different layers of an application.

In this chapter, we will discuss the command design pattern and how it can be utilized to encapsulate a request in the form of an object to perform it later or multiple times. We will learn about separating the business logic from the interface and how it helps to reuse each of them.

Structure

This chapter covers the following topics:

- Command design pattern
- Implementing the command pattern
- Summary
- Advantages and drawbacks
- Usage

Objectives

By the end of this chapter, the readers will learn about the command design pattern that allows you to create standalone objects for processing specific actions as and when required by the user. This chapter will also explore the benefits of decoupling the class that performs some operation from the one that invokes it, and discuss the implementation details of the command design pattern, the benefits it provides, and a few drawbacks, along with some of the places where this design pattern can be used.

Command design pattern

The command design pattern is a behavioral design pattern that allows encapsulating a request as an object, parameterizing other objects with different requests, delaying or queuing requests, and supporting undoable operations. It decouples the request invoker from the request operator and allows for the formation of a common interface that can be used across applications to perform actions on the operator.

Using the command design pattern, a class can delegate a request to a command object instead of implementing it on its own. A hard-wired request, on the other hand, couples the implementing class to a specific request and its operator, disallowing reusability and effortless integration. *Figure 17.2* exhibits the flow of operations in the command design pattern:

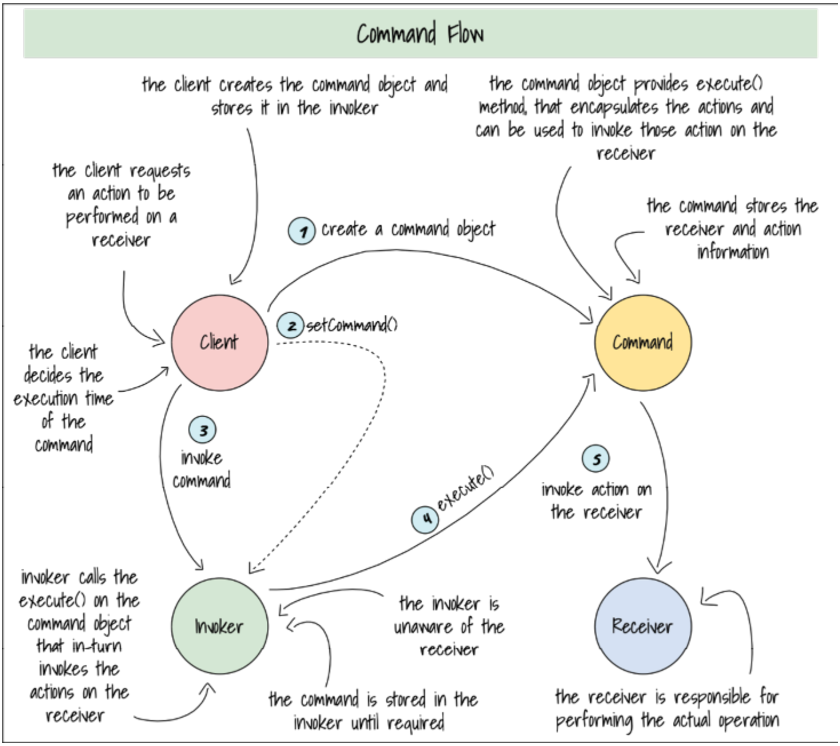

Command Flow
the client creates the command object and stores it in the invoker
the command object provides execute() method, that encapsulates the actions and can be used to invoke those action on the receiver
the client requests an action to be performed on a receiver
1 create a command object
the command stores the receiver and action information
Client
2 setCommand()
Command
the client decides the execution time of the command
3 invoke command
4 execute()
5 invoke action on the receiver
invoker calls the execute() on the command object that in-turn invokes the actions on the receiver
Invoker
the invoker is unaware of the receiver
Receiver
the command is stored in the invoker until required
the receiver is responsible for performing the actual operation

***Figure 17.2**: Command flow*

The client instantiates a command object, stores it in an invoker, and executes it when required to request an action on the receiver. The command object stores a reference to the receiver and the action that must be performed on it. The receiver performs the action when the command is executed by the invoker on the direction of the client. By using the command object, both the invoker and receiver remain unaware of each other's existence. The invoker can execute different types of commands without the need for separate integrations, and the receiver can perform the actions mentioned in the command without any knowledge of its invoker or client. *Figure 17.3* exhibits the architecture of the command design pattern:

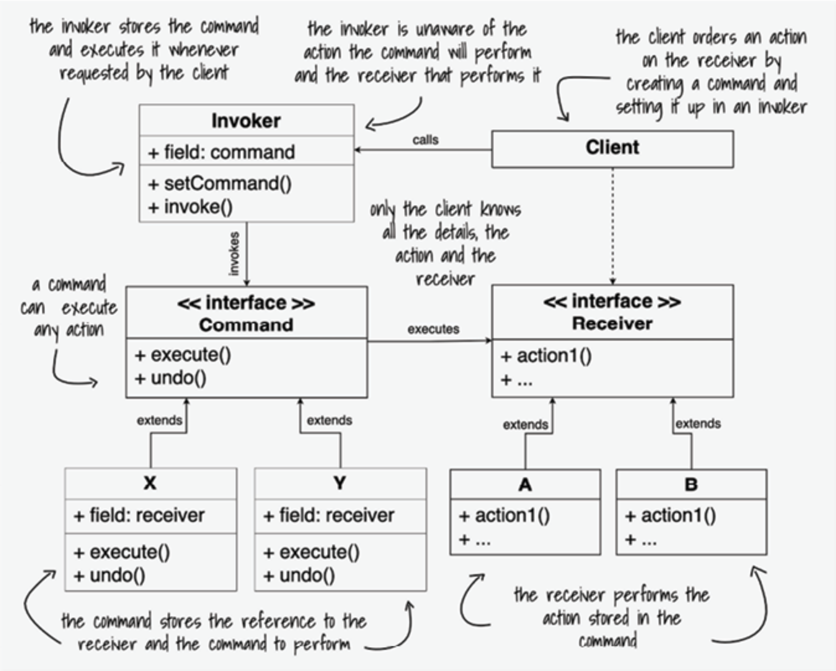

the invoker stores the command and executes it whenever requested by the client
the invoker is unaware of the action the command will perform and the receiver that performs it
the client orders an action on the receiver by creating a command and setting it up in an invoker
Invoker
+ field: command
+ setCommand()
+ invoke()
calls
Client
only the client knows all the details, the action and the receiver
invokes
a command can execute any action
<< interface >>
Command
+ execute()
+ undo()
executes
<< interface >>
Receiver
+ action1()
+ ...
extends
extends
extends
extends
X
+ field: receiver
+ execute()
+ undo()
Y
+ field: receiver
+ execute()
+ undo()
A
+ action1()
+ ...
B
+ action1()
+ ...
the command stores the reference to the receiver and the command to perform
the receiver performs the action stored in the command

***Figure 17.3**: Command design pattern*

The **Invoker** provides a setter method to set the command that the invoker instance should execute when the invoke method is called on it. The command interface provides two methods, which are **execute** and **undo**. The first is used to perform some action(s) on the receiver, and the second can be used to reverse those action(s). The receiver interface provides the set of operations that can be performed when the command is executed.

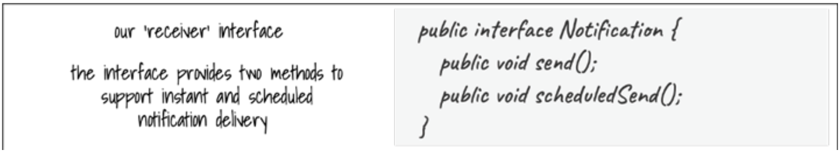
Implementing the command pattern

The example is covered in the entire chapter and discusses the notification service that can be used to send notifications to a user through email or messages. The example sheds light on the use of the command pattern to generalize architecture and streamline how various layers interact with each other.

We will learn about implementing the command design pattern and how it can be used to perform actions on a receiver by keeping our design loosely coupled. Let us start with the **Receiver** interface that lists the operations that can be performed, then shift our focus to the **Command** interface and the **Invoker**, and finally to the **Client** that creates and invokes a command.

Receiver

Receivers in the command design pattern are responsible for performing the actual operations. They are also known as operators. To which receiver a command interacts with is decided by the client. A receiver can be implemented to perform multiple actions, and commands can execute one or many of those. Different commands can execute different actions on the same receiver. *Figure 17.4* exhibits a **receiver** interface that provides two methods to support instant and scheduled delivery of notifications to the users:



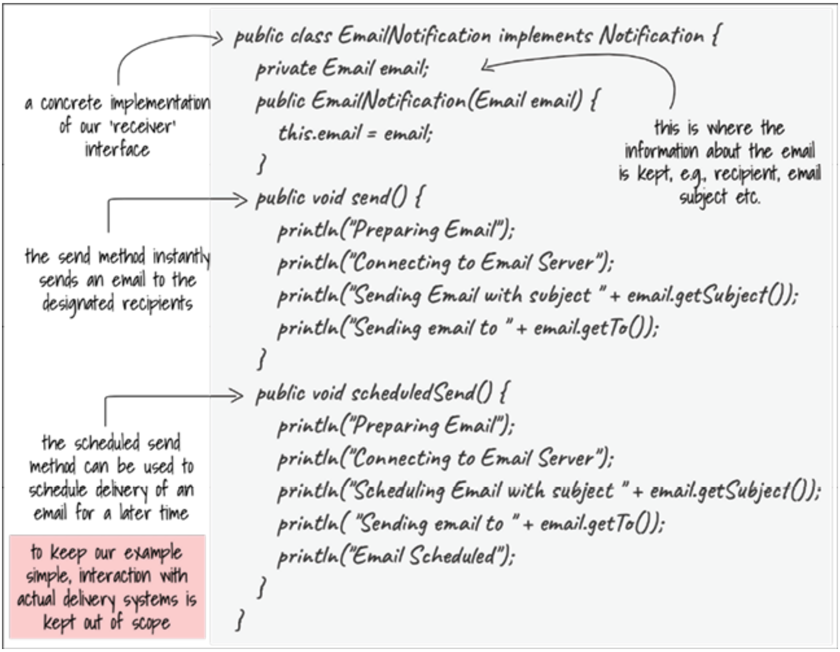
our 'receiver' interface

the interface provides two methods to support instant and scheduled notification delivery

```
public interface Notification {
   public void send();
   public void scheduledSend();
}
```

Figure 17.4: *Receiver—Interface notification*

The receivers do not necessarily need to be implemented in the form of interfaces; a command could store a reference to a class as well, though it is advisable to interact with interfaces and not directly with classes to keep the interaction loosely coupled. An interface allows interaction with multiple implementations of itself via polymorphism, something that cannot be done if the interaction is done directly with a concrete class implementation. *Figure 17.5* exhibits one of such concrete implementations. It stores the e-mail-related information in another object and overrides the two interface methods to support instant and scheduled delivery of notifications in the form of e-mails:



```
public class EmailNotification implements Notification {
    private Email email;
    public EmailNotification(Email email) {
        this.email = email;
    }
    public void send() {
        println("Preparing Email");
        println("Connecting to Email Server");
        println("Sending Email with subject " + email.getSubject());
        println("Sending email to " + email.getTo());
    }
    public void scheduledSend() {
        println("Preparing Email");
        println("Connecting to Email Server");
        println("Scheduling Email with subject " + email.getSubject());
        println( "Sending email to " + email.getTo());
        println("Email Scheduled");
    }
}
```

a concrete implementation of our 'receiver' interface

this is where the information about the email is kept, e.g. recipient, email subject etc.

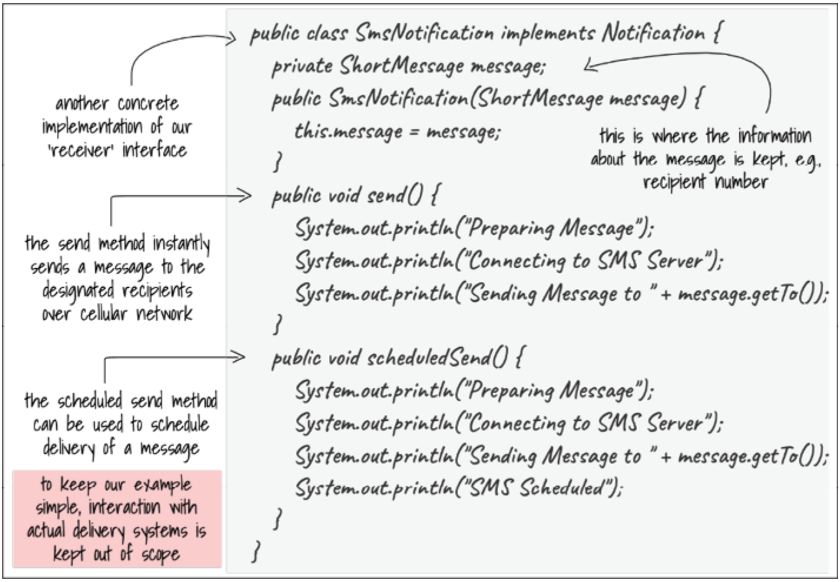
the send method instantly sends an email to the designated recipients

the scheduled send method can be used to schedule delivery of an email for a later time

to keep our example simple, interaction with actual delivery systems is kept out of scope

***Figure 17.5**: Receiver—E-mail notification*

Figure 17.6 exhibits another concrete implementation that delivers notifications in the form of short messages over a cellular network:



```
public class SmsNotification implements Notification {
    private ShortMessage message;
    public SmsNotification(ShortMessage message) {
        this.message = message;
    }
    public void send() {
        System.out.println("Preparing Message");
        System.out.println("Connecting to SMS Server");
        System.out.println("Sending Message to " + message.getTo());
    }
    public void scheduledSend() {
        System.out.println("Preparing Message");
        System.out.println("Connecting to SMS Server");
        System.out.println("Sending Message to " + message.getTo());
        System.out.println("SMS Scheduled");
    }
}
```

another concrete implementation of our 'receiver' interface

this is where the information about the message is kept, e.g. recipient number

the send method instantly sends a message to the designated recipients over cellular network

the scheduled send method can be used to schedule delivery of a message

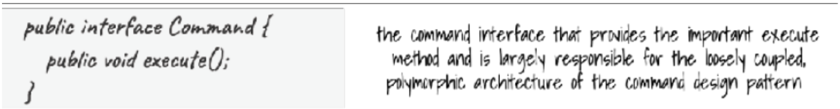
to keep our example simple, interaction with actual delivery systems is kept out of scope

Figure 17.6: Receiver—SMS notification

To keep our example simple and focus on the overall design, interaction with actual notification delivery systems is left to the user.

Command

A command is a representation of a stand-alone request. It stores all the information necessary to execute a command, that is, the action that is to be performed, the receiver that should perform it, and some other information, if required. *Figure 17.7* exhibits the command interface that is at the center of the command design pattern:

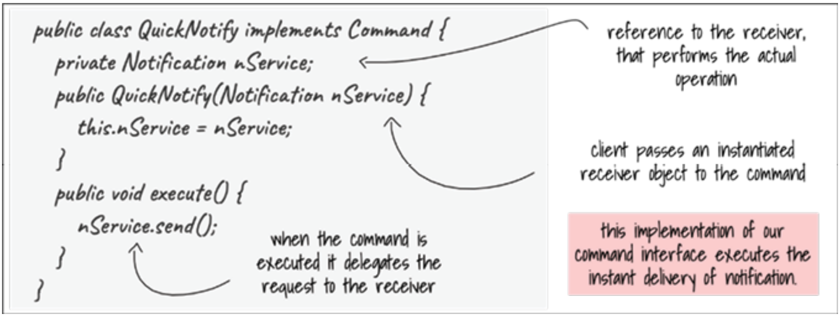


```
public interface Command {
   public void execute();
}
```

the command interface that provides the important execute method and is largely responsible for the loosely coupled, polymorphic architecture of the command design pattern

Figure 17.7: *Command interface*

Execution of a command has its own lifespan, independent of the original request. Commands can be grouped together to be processed in a batch or could be scheduled for delayed operation. They can be joined in a sequence to perform operations that depend on each other or can be stacked to perform undo operations. *Figure 17.8* exhibits a concrete implementation of our command interface:



```
public class QuickNotify implements Command {
   private Notification nService;
   public QuickNotify(Notification nService) {
      this.nService = nService;
   }
   public void execute() {
      nService.send();
   }
}
```

reference to the receiver, that performs the actual operation

client passes an instantiated receiver object to the command

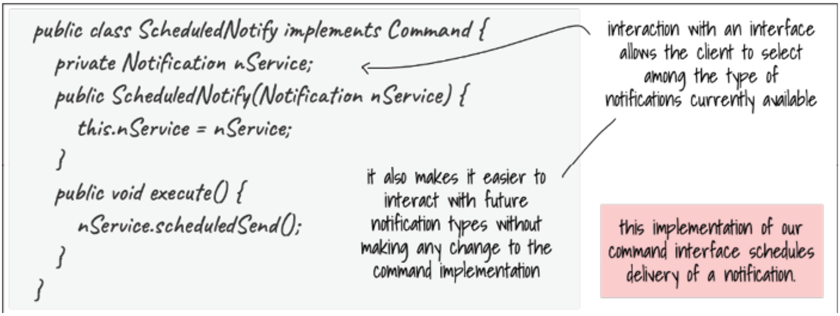
when the command is executed it delegates the request to the receiver

this implementation of our command interface executes the instant delivery of notification

Figure 17.8: *Command—Quick notify*

The implementation has a reference to a receiver that delegates the request for performing an action when the command is executed.

Figure 17.9 exhibits another concrete implementation of our **Command** interface. This implementation schedules the delivery of a notification, be it an email notification or a message over a cellular network.



```
public class ScheduledNotify implements Command {
   private Notification nService;
   public ScheduledNotify(Notification nService) {
      this.nService = nService;
   }
   public void execute() {
      nService.scheduledSend();
   }
}
```

interaction with an interface allows the client to select among the type of notifications currently available

it also makes it easier to interact with future notification types without making any change to the command implementation

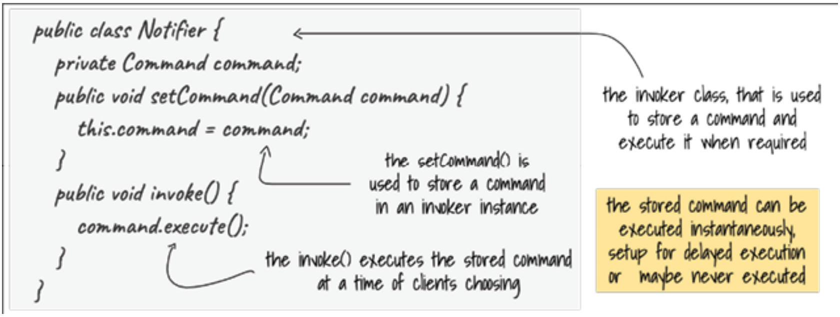
this implementation of our command interface schedules delivery of a notification.

***Figure 17.9**: Scheduled notify command*

Command objects allow for the construction of a common interface that can be utilized to delegate, sequence, or execute operations. The command can be executed at any time as required by the client. The same command can be executed multiple times by the client once it is stored in an invoker. A command object can be constructed anonymously to perform actions that are supposed to be performed only once. It is this object that generalizes the solution and binds the receiver and the invoker without them being aware of each other.

Invoker

The invoker invokes the command by calling the execute method. It can be used to log access-related information, perform an audit trail, or control execution of the command. The invoker can execute any command at any time chosen by the client. *Figure 17.10* exhibits an example of the invoker class that has a reference to the command object and provides a method to execute it:



```
public class Notifier {
  private Command command;
  public void setCommand(Command command) {
    this.command = command;
  }
  public void invoke() {
    command.execute();
  }
}
```

the invoker class, that is used to store a command and execute it when required

the setCommand() is used to store a command in an invoker instance

the invoke() executes the stored command at a time of clients choosing

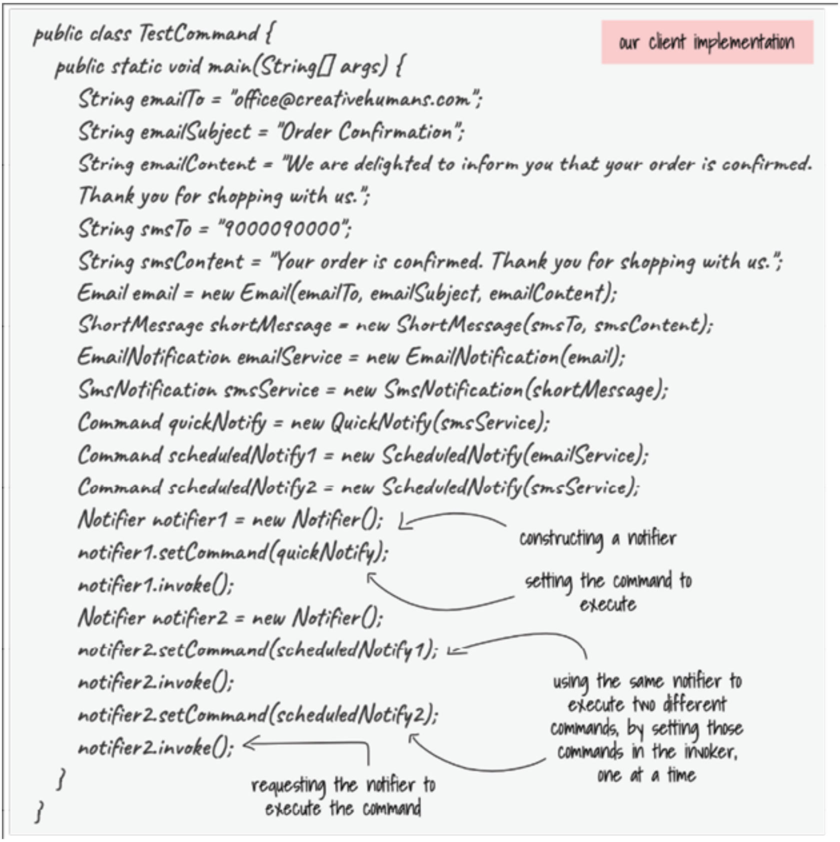
the stored command can be executed instantaneously, setup for delayed execution or maybe never executed

***Figure 17.10**: Invoker command*

The command can be executed at once, set up for a delayed execution, or may never be executed. The command set in the invoker can be changed at any time without any behavioral change in the invoker.

Client

The client is responsible for the creation of the command object by selecting a receiver and an action that the receiver can perform. *Figure 17.11* exhibits the implementation of a client process in the form a test class:



```
public class TestCommand {
   public static void main(String[] args) {
      String emailTo = "office@creativehumans.com";
      String emailSubject = "Order Confirmation";
      String emailContent = "We are delighted to inform you that your order is confirmed.
      Thank you for shopping with us.";
      String smsTo = "9000090000";
      String smsContent = "Your order is confirmed. Thank you for shopping with us.";
      Email email = new Email(emailTo, emailSubject, emailContent);
      ShortMessage shortMessage = new ShortMessage(smsTo, smsContent);
      EmailNotification emailService = new EmailNotification(email);
      SmsNotification smsService = new SmsNotification(shortMessage);
      Command quickNotify = new QuickNotify(smsService);
      Command scheduledNotify1 = new ScheduledNotify(emailService);
      Command scheduledNotify2 = new ScheduledNotify(smsService);
      Notifier notifier1 = new Notifier();
      notifier1.setCommand(quickNotify);
      notifier1.invoke();
      Notifier notifier2 = new Notifier();
      notifier2.setCommand(scheduledNotify1);
      notifier2.invoke();
      notifier2.setCommand(scheduledNotify2);
      notifier2.invoke();
   }
}
```

our client implementation

constructing a notifier

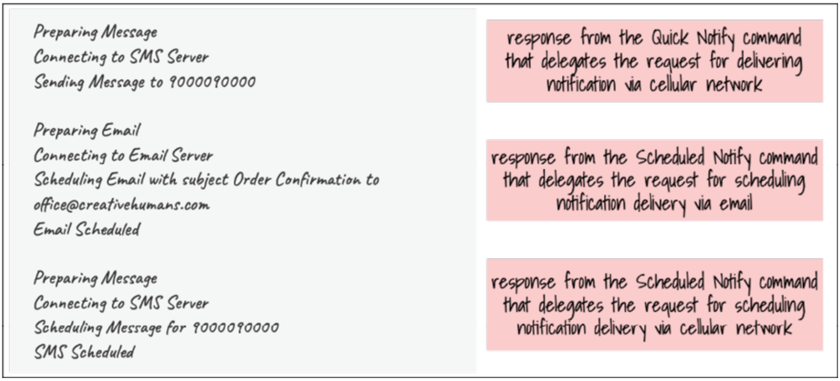
setting the command to execute

using the same notifier to execute two different commands, by setting those commands in the invoker, one at a time

requesting the notifier to execute the command

***Figure 17.11**: Client*

The client constructs the command objects and stores the necessary information in it, then passes this command to a newly constructed invoker object before finally making a call to that invoker object to execute the command. *Figure 17.12* demonstrates the result of executing the commands using the command design pattern:



```
Preparing Message
Connecting to SMS Server
Sending Message to 9000090000

Preparing Email
Connecting to Email Server
Scheduling Email with subject Order Confirmation to
office@creativehumans.com
Email Scheduled

Preparing Message
Connecting to SMS Server
Scheduling Message for 9000090000
SMS Scheduled
```

response from the Quick Notify command that delegates the request for delivering notification via cellular network

response from the Scheduled Notify command that delegates the request for scheduling notification delivery via email

response from the Scheduled Notify command that delegates the request for scheduling notification delivery via cellular network

***Figure 17.12**: Testing the command pattern*

Summary

The command design pattern allows for constructing standalone commands that can be triggered as and when required. These commands contain information about the action to perform and the receiver that executes it. The command pattern works with three main objects: the receiver, the command, and the invoker. The command design pattern promotes separation of concern of and a loosely coupled architecture.

The command pattern can also expose another method called undo. The **undo** method can be called from an invoker to reverse the previous operation performed on some command. The undo operation is very useful when we are dealing with actions that are reversible, for example, maintaining binary switches to control information flow or traffic on a system. Together, the **execute** and **undo** methods can be used to perform operations that rely on a sequence, as in a stack, for example, controlling log behavior by increasing and decreasing the severity level of the information allowed for logging from within the execute and undo methods. *Figure 17.13* exhibits a high-level view of the command design pattern:

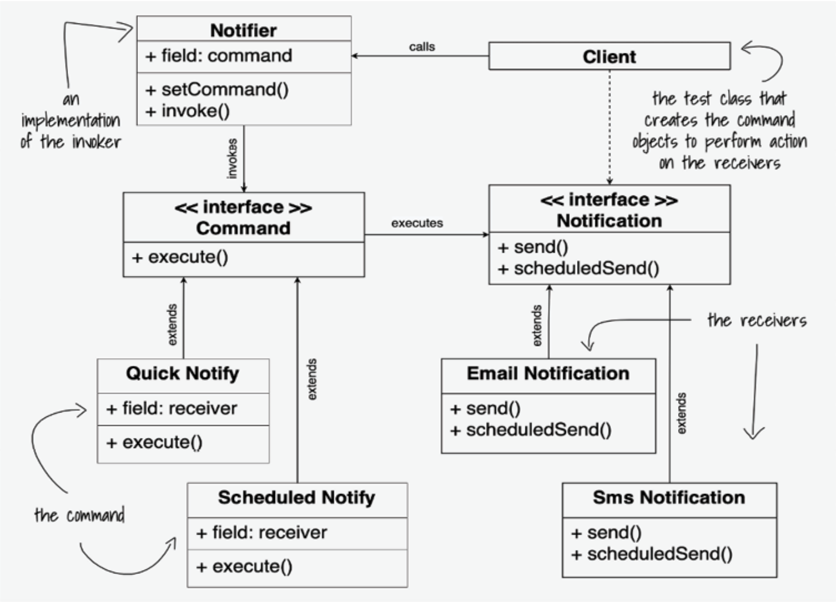

Notifier
+ field: command
+ setCommand()
+ invoke()
calls
Client
an implementation of the invoker
the test class that creates the command objects to perform action on the receivers
invokes
<< interface >> Command
+ execute()
executes
<< interface >> Notification
+ send()
+ scheduledSend()
extends
extends
the receivers
Quick Notify
+ field: receiver
+ execute()
extends
Email Notification
+ send()
+ scheduledSend()
extends
the command
Scheduled Notify
+ field: receiver
+ execute()
Sms Notification
+ send()
+ scheduledSend()

***Figure 17.13**: Command design pattern—Defined*

It presents the classes that we have used in our representation of the command design pattern. In our example, we have concrete implementations of the **notification** interface that resemble receivers, two command objects that call the methods exposed by the notification interface, and an invoker that executes the command objects.

Advantages and drawbacks

The following list outlines the key benefits of the command design pattern:

- It allows to encapsulate the request in an object.
- It promotes separation of concerns by utilizing a layered approach of software development where no two layers are tightly coupled with each other.
- It is easy to introduce new commands without changing the existing logic of the application.
- It can also be used to define reversal operations.

The drawbacks of the command method design pattern are:

- Multiple command implementations can become difficult to manage, especially when they are not reusable.
- A lot of code must be written and managed for the command design pattern.
- The execute method in the command interface does not allow returning a value in its original definition.

Usage

The following list outlines some of the scenarios that are relevant to the command design pattern:

- When application restructuring is required to reduce coupling between objects without any major changes in the underlying logic.
- When the dynamic behavior of the application allows it to perform distinct actions using the same set of operations.

Conclusion

Command design pattern presents a reliable approach for defining commands at runtime, that can perform all sorts of actions while reducing object coupling and streamlining the integration mechanism. The three types of objects that are necessary for the proper functioning of the command design pattern are the command, the receiver, and the invoker. A command stores the information about the action and its executor, and the invoker executes it.

In the next chapter, we will learn about the strategy design pattern and how it can be utilized to switch between different algorithms in runtime.

Questions

In this section, we will sharpen our brains by indulging in some fun brain activities. Let us get started.

Find the odd one out

1. Invoker, command, executor, receiver
2. Separation of concern, loose coupling, layered architecture, predefined commands
3. Series of operations, redo, undo, operation stack
4. Command design pattern:
    a. Allows creation of standalone commands
    b. Is a behavioral design pattern
    c. Reduces the need for object interaction
    d. Reduces coupling between objects
2. Types of classes used in the command design pattern are:
    a. Exporter
    b. Caller
    c. Invoker
    d. Converter
3. Operations performed via the command design pattern are:
    a. Queueing of operations in a predefined order
    b. Reversal of an operation using undo
    c. Callback operations
    d. Execution of the same operation multiple times

Select two correct answers

1. State design pattern:
    a. Employs polymorphism
    b. Increases application complexity
    c. Promotes separation of concern
    d. Increases coupling
2. The command object provides these methods:
    a. Execute
    b. Perform
    c. Redo
    d. Undo

3. What goes together:
    a. Encapsulation, command object, action
    b. Structural, command design pattern, tight coupling
    c. Command design pattern, behavioral, loose coupling
    d. Invoker, commander, receiver

Answers

This section provides answers to the given questions.

Find the odd one out

1. Executor
2. Predefined commands
3. Operation stack
4. Reduces the need for object interaction
5. Invoker
6. Callback operations

Select two correct answers

1. Employs polymorphism, promotes separation of concerns
2. Execute, undo
3. Encapsulation, command object, action, and command design pattern, behavioral, loose coupling

Join our Discord space

Join our Discord workspace for latest updates, offers, tech happenings around the world, new releases, and sessions with the authors:

https://discord.bpbonline.com

CHAPTER 18

Strategy

Introduction

While writing software applications, it is often the case that the control flow branches out to different paths based on the use case. The request endpoint may have been developed in a way that it can work for varying requirements, e.g., a request to store a record can be fulfilled by storing it in a database, an in-memory cache, or a file. The initial steps of the request can be the same, but the storage process can be implemented in three distinct ways. These three options can be thought of as strategies that can be implemented and used to dynamically alter the control flow, as shown in the following figure:

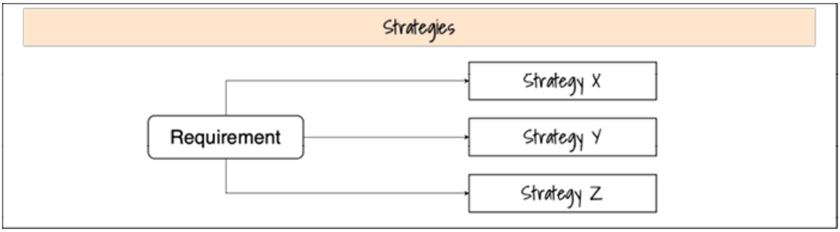


***Figure 18.1**: Varying strategies*

In this chapter, we will learn about strategies and how they can be used to alter the behavior of a process at runtime.

Structure

This chapter covers the following topics:

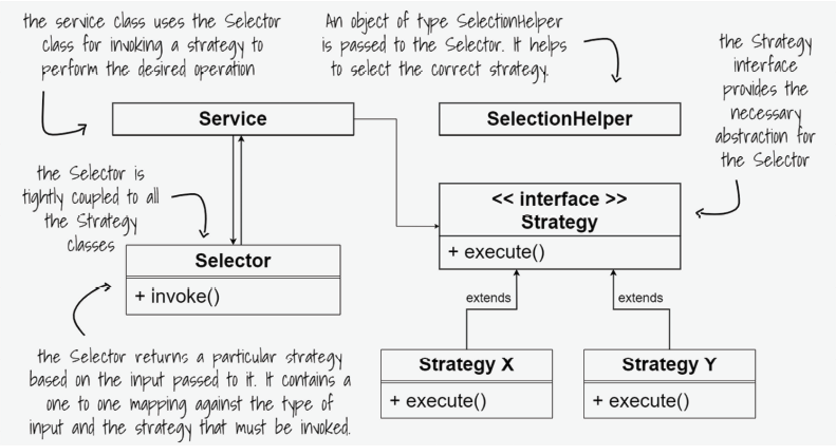
- Strategy design pattern
- Implementing the strategy pattern
- Summary
- Advantages and drawbacks
- Usage

Objectives

By the end of this chapter, the readers will learn about the strategy design pattern that allows us to develop strategies and use them interchangeably. These strategies can be selected at runtime to alter the course of action and perform distinct operations, all while reusing the same codebase. They will also be able to understand the workings of the strategy design pattern and how it can be used to employ various strategies with ease and efficiency.

Strategy design pattern

The strategy design pattern is a behavioral design pattern that segregates algorithms into their own classes and allows them to be used interchangeably, as strategies:



the service class uses the Selector class for invoking a strategy to perform the desired operation
An object of type SelectionHelper is passed to the Selector. It helps to select the correct strategy.
the Strategy interface provides the necessary abstraction for the Selector
Service
SelectionHelper
the Selector is tightly coupled to all the Strategy classes
<< interface >>
Strategy
+ execute()
Selector
+ invoke()
extends
extends
Strategy X
+ execute()
Strategy Y
+ execute()
the Selector returns a particular strategy based on the input passed to it. It contains a one to one mapping against the type of input and the strategy that must be invoked.

***Figure 18.2**: Strategy design pattern*

This pattern advocates that a class that can solve a problem in multiple ways can be refactored and its algorithms can be extracted into separate classes. Transposable algorithms are put into classes that implement the same interface and can be interchanged at runtime using polymorphism. This allows for segregating and encapsulating the logic into classes, enabling reuse and easy maintenance. These classes can then be used as strategies. This pattern is one of the simplest yet most powerful examples of polymorphism in action.

It promotes the principle of open for extension and allows adding new strategies without affecting the existing codebase. This provides for a plug-and-play architecture that is extensible yet manageable.

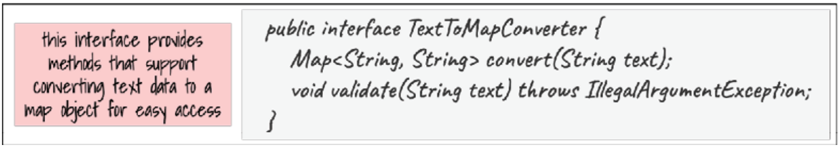
Implementing the strategy pattern

The example discussed in this chapter is about a conversion service that can be used to convert information stored in a serializable form, e.g., XML, into a map. This example uses the strategy design pattern to segregate the conversion logic into multiple classes, each performing a specific type of conversion.

We will learn about the implementation of the strategy design pattern and how it can be used to dynamically choose the correct strategy at runtime.

Strategy interface

The strategy design pattern utilizes polymorphism to access and execute different strategies. A group of strategies implemented as classes must implement the same interface for the design pattern to work. *Figure 18.3* exhibits an interface that defines the set of behaviors that must be exposed by interchangeable strategies:



this interface provides methods that support converting text data to a map object for easy access

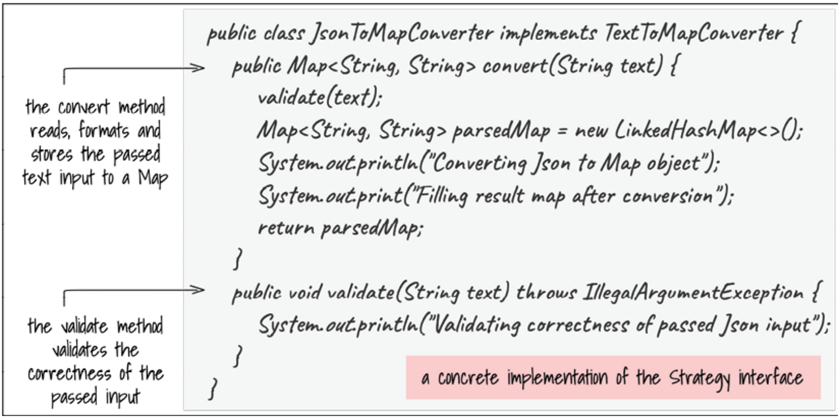
```
public interface TextToMapConverter {
    Map<String, String> convert(String text);
    void validate(String text) throws IllegalArgumentException;
}
```

***Figure 18.3**: The strategy interface*

The interface binds all the strategies together and provides the means to switch between strategies at runtime.

Strategies

A strategy is a method to approach a problem. It is a specific procedure that, when executed successfully, results in a desired outcome. An approach can be materialized in the form of an algorithm and implemented as a class to carve out a strategy. *Figure 18.4* exhibits a concrete implementation of the strategy interface:



```
public class JsonToMapConverter implements TextToMapConverter {
    public Map<String, String> convert(String text) {
        validate(text);
        Map<String, String> parsedMap = new LinkedHashMap<>();
        System.out.println("Converting Json to Map object");
        System.out.print("Filling result map after conversion");
        return parsedMap;
    }
    public void validate(String text) throws IllegalArgumentException {
        System.out.println("Validating correctness of passed Json input");
    }
}
```

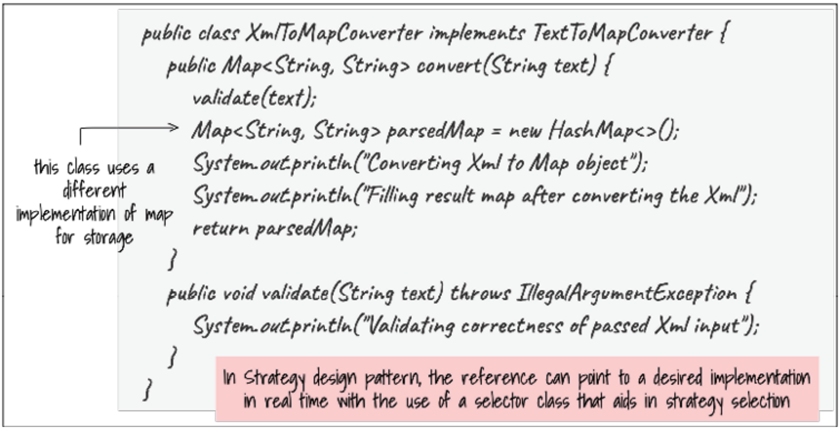
the convert method reads, formats and stores the passed text input to a Map

the validate method validates the correctness of the passed input

a concrete implementation of the Strategy interface

***Figure 18.4**: Strategy implementation—Json*

The preceding strategy parses the **Json** input, converts it into key-value pairs, and stores it into a map. *Figure 18.5* exhibits a similar strategy:



```
public class XmlToMapConverter implements TextToMapConverter {
   public Map<String, String> convert(String text) {
      validate(text);
      Map<String, String> parsedMap = new HashMap<>();
      System.out.println("Converting Xml to Map object");
      System.out.println("Filling result map after converting the Xml");
      return parsedMap;
   }
   public void validate(String text) throws IllegalArgumentException {
      System.out.println("Validating correctness of passed Xml input");
   }
}
```

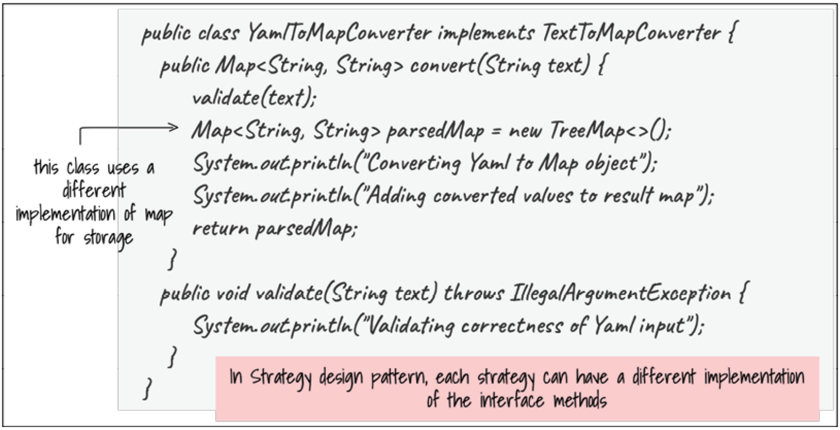
this class uses a different implementation of map for storage

In Strategy design pattern, the reference can point to a desired implementation in real time with the use of a selector class that aids in strategy selection

***Figure 18.5**: Strategy implementation—Xml*

This strategy performs the task of converting text to a map object, but when the input is in **Xml** format. Strategies that execute similar operations can be considered as a group of strategies. They have varying behaviors but have the same end goal.

Figure 18.6 exhibits another such strategy that takes a **Yaml** input, converts and stores it into a **map** object:



```
public class YamlToMapConverter implements TextToMapConverter {
  public Map<String, String> convert(String text) {
      validate(text);
      Map<String, String> parsedMap = new TreeMap<>();
      System.out.println("Converting Yaml to Map object");
      System.out.println("Adding converted values to result map");
      return parsedMap;
   }
  public void validate(String text) throws IllegalArgumentException {
      System.out.println("Validating correctness of Yaml input");
   }
}
```

this class uses a different implementation of map for storage

In Strategy design pattern, each strategy can have a different implementation of the interface methods

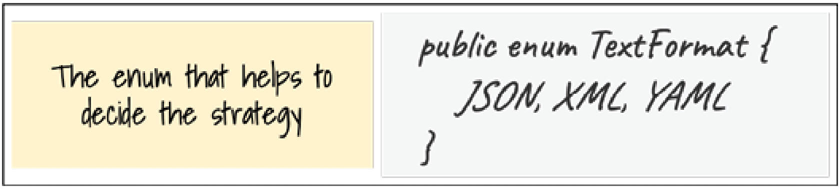
***Figure 18.6**: Strategy implementation—Yaml*

All three strategies have the same goal: to convert text input to a map object. Their behavior and execution process are slightly different and depend on the algorithm that they implement. By encapsulating strategies into classes and hierarchies, they can be used interchangeably to perform operations with a varying set of behaviors. They are reusable and promote code deduplication.

Selection of strategies

Selecting the correct strategy for execution is as important as having the strategies themselves. In a strategy design pattern, the selection and invocation logic are encapsulated in a class that acts as a bridge between the strategy and the actual invoker. It provides the necessary abstraction layer for the invoker to remain loosely coupled with the strategies.

The selection of the strategy is often decided based on the input parameters passed to this class. To help with the selection process, we can use any type and number of parameters. We will make use of an enum class, in the example that we are following, to select the correct strategy. *Figure 18.7* exhibits a simple enum class:

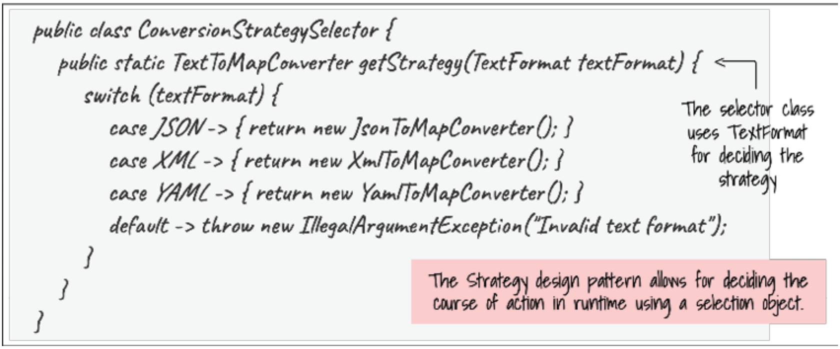


The enum that helps to decide the strategy

```
public enum TextFormat {
    JSON, XML, YAML
}
```

***Figure 18.7**: Strategy selection helper*

Next, we will look at the class that is responsible for deciding the correct strategy. *Figure 18.8* exhibits a simple strategy selector that takes an enum object as input and uses it to select the correct strategy:



```
public class ConversionStrategySelector {
    public static TextToMapConverter getStrategy(TextFormat textFormat) {
        switch (textFormat) {
            case JSON -> { return new JsonToMapConverter(); }
            case XML -> { return new XmlToMapConverter(); }
            case YAML -> { return new YamlToMapConverter(); }
            default -> throw new IllegalArgumentException("Invalid text format");
        }
    }
}
```

The selector class uses TextFormat for deciding the strategy

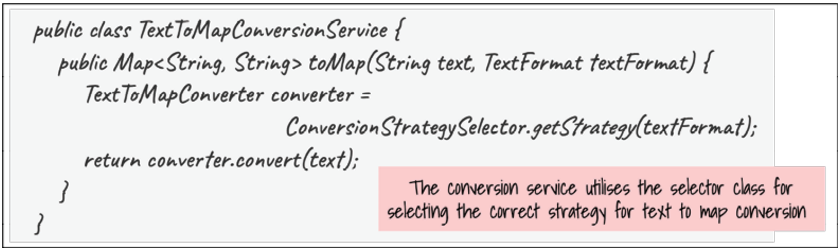
The Strategy design pattern allows for deciding the course of action in runtime using a selection object.

***Figure 18.8**: Strategy selector*

The selector class is tightly coupled to the strategies and therefore has a direct impact when a strategy is added, replaced, or deleted. This class must always be in sync with the strategies to ensure correct behavior.

Invoking strategies

With the selection logic in place, the strategies can now be invoked by using the selector class to fetch the correct strategy and then executing it. The service class exhibited in *Figure 18.9* makes use of the strategy selector class to fetch the correct strategy implementation by passing the enum object to it:



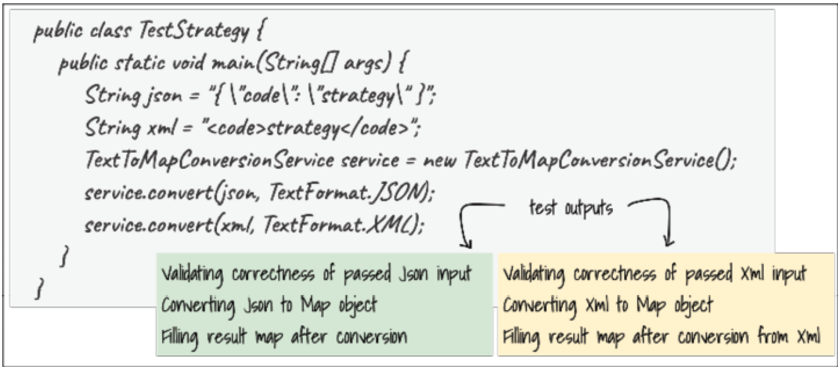
```
public class TextToMapConversionService {
    public Map<String, String> toMap(String text, TextFormat textFormat) {
        TextToMapConverter converter =
                                    ConversionStrategySelector.getStrategy(textFormat);
        return converter.convert(text);
    }
}
```

The conversion service utilises the selector class for selecting the correct strategy for text to map conversion

***Figure 18.9**: Conversion service*

Testing the strategies

We have written a test class to verify the working of the strategy design pattern. Let us execute the same as shown in *Figure 18.10*:



```
public class TestStrategy {
    public static void main(String[] args) {
        String json = "{ \"code\": \"strategy\" }";
        String xml = "<code>strategy</code>";
        TextToMapConversionService service = new TextToMapConversionService();
        service.convert(json, TextFormat.JSON);
        service.convert(xml, TextFormat.XML);
    }
}
```

test outputs

```
Validating correctness of passed Json input
Converting Json to Map object
Filling result map after conversion
```

```
Validating correctness of passed Xml input
Converting Xml to Map object
Filling result map after conversion from Xml
```

***Figure 18.10**: Testing the strategies*

In the preceding example, the conversion service is called twice with a different set of inputs, first for converting a **Json** string and then for converting an **Xml** string. The enum value passed as input allowed for the selection of the correct converter object, printing the correct output to validate the working of the strategy design pattern.

Summary

The strategy design pattern aims to segregate and encapsulate behaviors into classes. It utilizes polymorphism for selecting the correct behavior. It effectively replaces the conditional checks for behavior selection and provides a systematic and reusable approach instead. It separates the selection logic from the context in which the strategies are used.

A strategy selector, a class that encapsulates the selection logic, and a few strategies are the three core elements of the strategy design pattern. They are required for the implementation of the strategy pattern. In *Figure 18.11*, the strategy design pattern is reimaged with the help of the classes that we have used in the example:

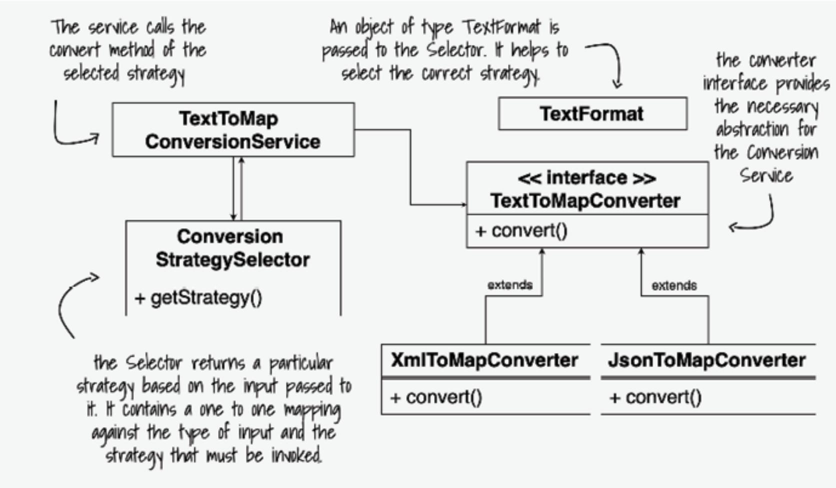

The service calls the convert method of the selected strategy
An object of type TextFormat is passed to the Selector. It helps to select the correct strategy
the converter interface provides the necessary abstraction for the Conversion Service
TextToMap ConversionService
TextFormat
<< interface >> TextToMapConverter
+ convert()
Conversion StrategySelector
+ getStrategy()
extends
extends
XmlToMapConverter
+ convert()
JsonToMapConverter
+ convert()
the Selector returns a particular strategy based on the input passed to it. It contains a one to one mapping against the type of input and the strategy that must be invoked

Figure 18.11*: Strategy design pattern—Defined*

Advantages and drawbacks

Refer to the following list, which outlines the benefits of the strategy design pattern:

- It allows for selecting and switching between algorithms at runtime.
- It promotes the principle of separation of concern by encapsulating strategies into their own classes.
- It encapsulates the selection logic into a single class, allowing for reusability.
- Addition of new strategies or removal of old ones is very straightforward and does not impact the surrounding code blocks.

The drawbacks of the strategy design pattern are:

- The class that hosts the selection logic is tightly coupled with all the strategies. It also acts as the abstraction layer between the strategies and the invoker, and therefore is always open to change.
- Too many strategies can lead to confusion and increased complexity associated with internal communication between classes.
- If the strategies do not implement the same interface, the selection logic can become complex and hard to maintain.

Usage

The following list outlines some scenarios that are relevant to the strategy design pattern:

- When there are multiple approaches to execute a task, and the need is to switch between them at runtime.
- To separate the selection of strategies from the context in which they are used.

Conclusion

The strategy design pattern presents a clean approach for segregating and encapsulating algorithms into their own classes. It supports principles like open to extension and separation of concern and allows the development and use of algorithms in isolation. It acts as an abstraction for the invoker layer to reduce coupling and ease maintenance.

In the next chapter, we will learn about the pipeline design pattern that allows us to execute operations in a sequential manner.

Questions

In this section, we will sharpen our brains by indulging in some fun brain activities. Let us get started.

Find the odd one out

1. Design principle not supported by the strategy design pattern:
    a. Open for extension
    b. Separation of concern
    c. Layered architecture
    d. Single responsibility
2. Strategy design pattern:
    a. Encapsulates algorithms into their own classes
    b. Promotes reusability
    c. Is a structural design pattern
    d. Allows for switching strategies at runtime
3. What is correct with respect to a strategy:
    a. Can be selected and switched at runtime
    b. Are methods of the same class
    c. Cannot be invoked directly
    d. Are tightly coupled to each other

Select two correct answers

1. Strategy design pattern:
    a. Supports the principle of open for extension
    b. Restricts reusability
    c. Supports the principle of separation of concerns

    d. Increases coupling
2. Strategies are:
    a. Non-extendable
    b. Reusable
    c. Immutable
    d. Interchangeable

Answers

This section provides the answers to the questions given.

Find the odd one out

1. Layered architecture
2. Is a structural design pattern
3. Can be selected and switched in runtime

Select two correct answers

1. Supports the principle of open for extension, and supports the principle of separation of concern
2. Reusable and interchangeable

Join our Discord space
Join our Discord workspace for latest updates, offers, tech happenings around the world, new releases, and sessions with the authors:

https://discord.bpbonline.com

CHAPTER 19

Pipeline

Introduction

In software development, when an operation becomes too large or complex, it is often advised to split it, creating interconnected sub-tasks that allow for reducing complexity and increasing maintainability. These subtasks would often require being executed in an orderly fashion to complete the operation. For example, the process to generate a report after some complex computation can be split into multiple sub-tasks like fetching data from the database, massaging the data, and then preparing the report, as illustrated in the following figure:

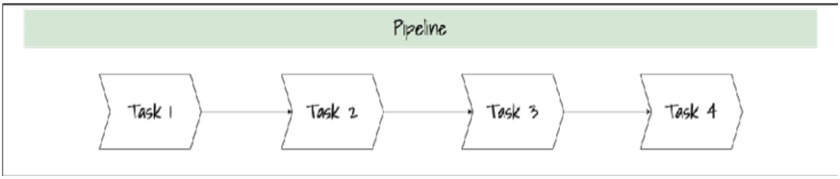


***Figure 19.1**: Tasks in a pipeline*

These sub-tasks can be represented with their own classes and can be executed in the said sequence to generate the report. The underlying pattern that is formed by clubbing these tasks in a sequence is known as a pipeline. In this chapter, we will learn about pipelines and how they can be utilized to perform operations by combining tasks together and running them in a sequence.

Structure

This chapter covers the following topics:

- Pipeline design pattern
- Implementing the pipeline pattern
- Summary
- Advantages and drawbacks
- Usage

Objectives

By the end of this chapter, the readers will learn about the pipeline design pattern that allows joining tasks together in a sequence to execute an operation. It facilitates and promotes modular development and emphasizes code reuse. These tasks can be arranged in many ways to change the overall behavior of the pipeline.

After finishing this chapter, the readers will also be able to understand the working of the pipeline design pattern and how it can be used to break down large, complex tasks into smaller, manageable ones to increase operational efficiency.

Pipeline design pattern

The pipeline design pattern is a behavioral design pattern that allows for executing a task by breaking it down into multiple sub-tasks and running them in sequence, as shown in the following figure:

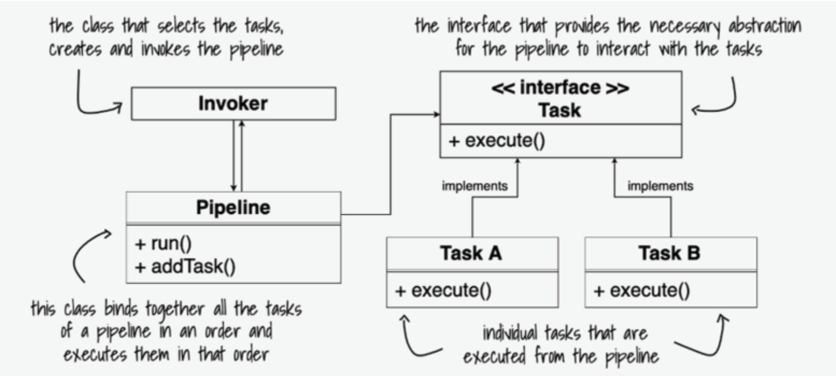

the class that selects the tasks, creates and invokes the pipeline
the interface that provides the necessary abstraction for the pipeline to interact with the tasks
Invoker
<< interface >>
Task
+ execute()
implements
implements
Pipeline
+ run()
+ addTask()
Task A
+ execute()
Task B
+ execute()
this class binds together all the tasks of a pipeline in an order and executes them in that order
individual tasks that are executed from the pipeline

***Figure 19.2**: Pipeline design pattern*

This pattern promotes reusability and extensibility. Once a task is defined, it can be reused in multiple pipelines. A pipeline can also be extended or shortened by adding or removing tasks from it, without impacting other tasks. For example, think of a pipeline that updates the status of user queries in bulk, but it does not have a way to fast-track the processing of open queries that are older than 48 hours.

To add this feature, a new task can be added to the pipeline that finds all the unanswered or open queries that were created more than 48 hours ago and adds them to a priority list for faster resolution. This new task adds to the existing behavior of the pipeline without impacting any of the earlier tasks and can also be reused later. *Figure 19.3* exhibits the addition of a task to the pipeline:

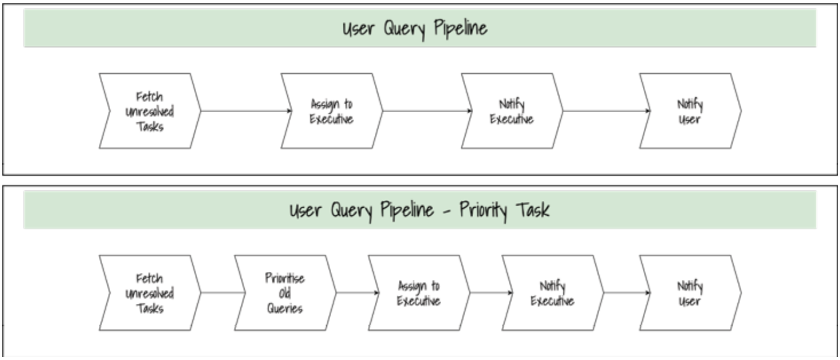

User Query Pipeline
Fetch Unresolved Tasks
Assign to Executive
Notify Executive
Notify User
User Query Pipeline - Priority Task
Fetch Unresolved Tasks
Prioritise Old Queries
Assign to Executive
Notify Executive
Notify User

***Figure 19.3**: User query pipeline*

Each task in a pipeline is like a distinct component that is represented by a class of its own. Tasks are usually decoupled and can be arranged in any order, unless they process the output of the previous task, in which case the order must be well defined.

For example, think of a pipeline that analyzes refund requests for an e-commerce firm and approves them for further processing. To verify and process a refund request, the pipeline should have interconnected tasks. One of the tasks could do the validation, and all the approved refunds can be added to a list of valid requests, which later can be consumed by another task that processes the refund back to the user's wallet. *Figure 19.4* exhibits the tasks that take the output of the previous task as an input:

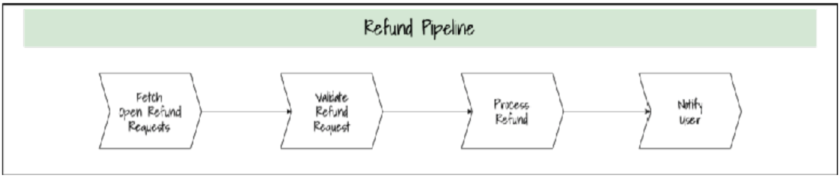


***Figure 19.4**: Refund pipeline*

In a pipeline in which individual tasks return an output, the output of one task becomes the input for the next, and therefore, the developer must ensure the compatibility between tasks that are supposed to run in sequence. If the tasks do not return an output, then they can be put in any order necessary.

This design allows for modular development as each task can be individually developed and used in a pipeline. As shown in *Figure 19.5*, pipelines can also be scaled to run tasks in parallel or in a distributed manner, often using a different processor, usually in a cloud infrastructure.

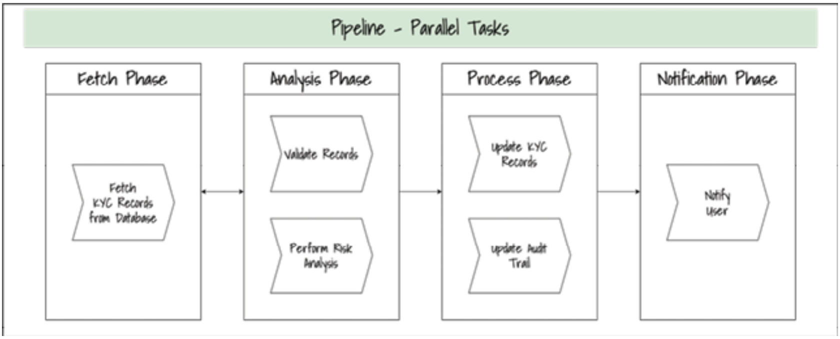

Pipeline - Parallel Tasks
Fetch Phase
Fetch KYC Records from Database
Analysis Phase
Validate Records
Perform Risk Analysis
Process Phase
Update KYC Records
Update Audit Trail
Notification Phase
Notify User

***Figure 19.5**: Pipeline having parallel tasks*

Implementing the pipeline pattern

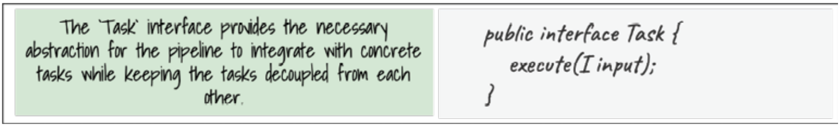
The example that we will walk you through in this chapter will explain the working of the pipeline pattern effectively. We will use the pipeline design pattern for reading the content of a text file and counting all the distinct words present in it.

The process of finding the distinct words will be broken down into small pieces and then clubbed together via a pipeline. Each task, except the first one, will consume the output of the previous task and will be placed in the pipeline in a specific order based on the functionality that they provide.

We will learn about the implementation of the pipeline design pattern and how it can be used to bind together distinct steps of a pipeline.

Task interface

The pipeline design pattern utilizes abstraction and polymorphism to bind distinct tasks together. Tasks must implement the same interface for the design pattern to work. *Figure 19.6* exhibits an interface that defines the behavior that must be implemented by these tasks:

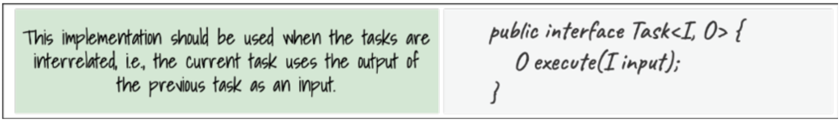


The `Task` interface provides the necessary abstraction for the pipeline to integrate with concrete tasks while keeping the tasks decoupled from each other.

```
public interface Task {
    execute(I input);
}
```

***Figure 19.6**: Task interface*

This is a generic adaptation of the *Task* interface that allows executing a task. *Figure 19.7* exhibits another definition of the *Task* interface that has a distinct input and output type, allowing each task to process different types of inputs. It also helps to validate that the previous and next tasks are not incompatible.



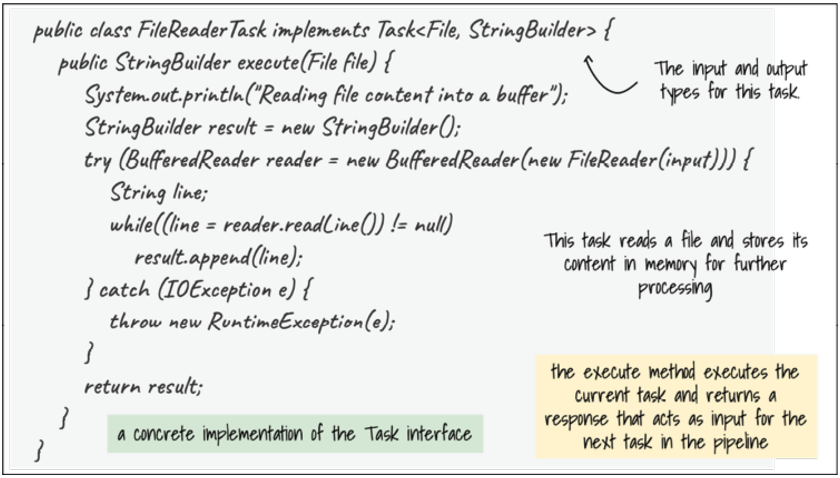
This implementation should be used when the tasks are interrelated, i.e., the current task uses the output of the previous task as an input.

```
public interface Task<I, O> {
    O execute(I input);
}
```

Figure 19.7*: Task Interface and the interrelated tasks*

Concrete tasks

Each concrete task is a separate component that acts as a building block of a pipeline. The pipeline executes these tasks in some pre-specified order to perform an operation. These concrete tasks implement the *Task* interface and provide a definition for the *execute* method. *Figure 19.8* exhibits a concrete implementation of the *Task* interface. The following task reads a file and stores its content in memory for further processing. It returns an object of type **StringBuilder** for consumption by the next task in the pipeline.



```
public class FileReaderTask implements Task<File, StringBuilder> {
    public StringBuilder execute(File file) {
        System.out.println(“Reading file content into a buffer”);
        StringBuilder result = new StringBuilder();
        try (BufferedReader reader = new BufferedReader(new FileReader(input))) {
            String line;
            while((line = reader.readLine()) != null)
                result.append(line);
        } catch (IOException e) {
            throw new RuntimeException(e);
        }
        return result;
    }
}
```

The input and output types for this task.

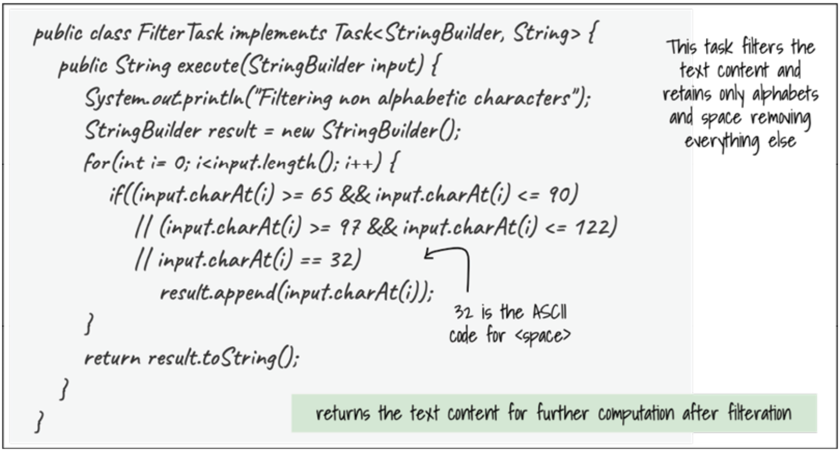
This task reads a file and stores its content in memory for further processing

the execute method executes the current task and returns a response that acts as input for the next task in the pipeline

a concrete implementation of the Task interface

Figure 19.8*: Task—File reader*

The tasks in the pipeline are decoupled from each other and only interact via the objects that they consume as input. The pipeline binds these tasks together. *Figure 19.9* exhibits another task implementation. It filters the content of the file that we read in the previous task, and retains only the alphabet and the space:



```
public class FilterTask implements Task<StringBuilder, String> {
   public String execute(StringBuilder input) {
      System.out.println("Filtering non alphabetic characters");
      StringBuilder result = new StringBuilder();
      for(int i= 0; i<input.length(); i++) {
         if((input.charAt(i) >= 65 && input.charAt(i) <= 90)
            || (input.charAt(i) >= 97 && input.charAt(i) <= 122)
            || input.charAt(i) == 32)
               result.append(input.charAt(i));
      }
      return result.toString();
   }
}
```

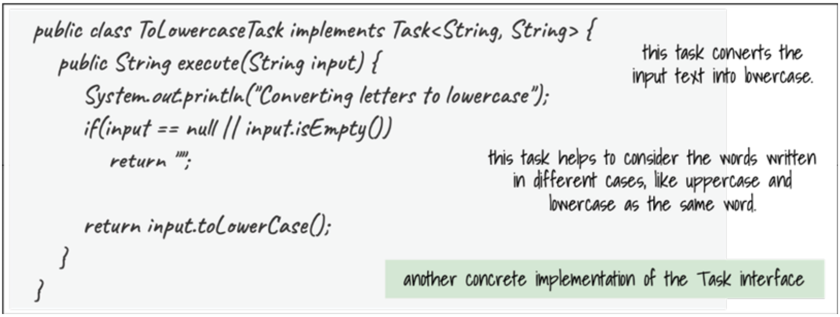
This task filters the text content and retains only alphabets and space removing everything else

32 is the ASCII code for <space>

returns the text content for further computation after filteration

***Figure 19.9**: Task: filter*

The next step in the pipeline, as exhibited in *Figure 19.10*, converts the entire text to lowercase. The file content might not always be in the same form, and therefore the same word, but written using a combination of lowercase and capital case letters, will be processed more than once. To avoid multiple counts of the same word, we can change the entire text to lowercase or any other case. This task can be omitted if the file's content is in the same form.



```
public class ToLowercaseTask implements Task<String, String> {
   public String execute(String input) {
      System.out.println("Converting letters to lowercase");
      if(input == null || input.isEmpty())
         return "";

      return input.toLowerCase();
   }
}
```

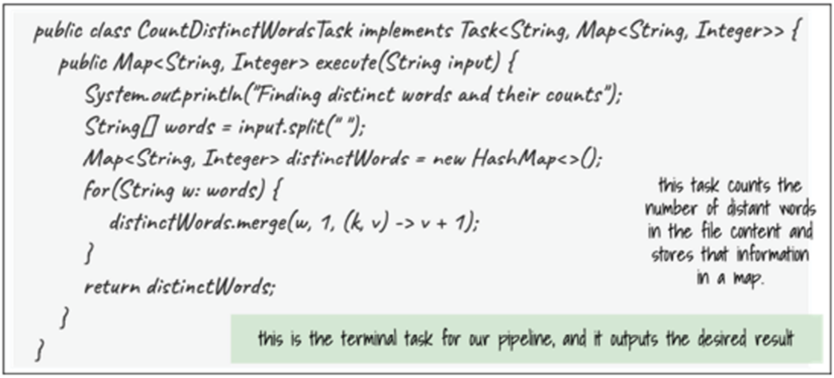
this task converts the input text into lowercase.

this task helps to consider the words written in different cases, like uppercase and lowercase as the same word.

another concrete implementation of the Task interface

***Figure 19.10**: Task: To lowercase*

The last step of the pipeline counts the frequency of words in the file content and stores it in a *Map* object. *Figure 19.11* exhibits the implementation of the final task:



```
public class CountDistinctWordsTask implements Task<String, Map<String, Integer>> {
    public Map<String, Integer> execute(String input) {
        System.out.println("Finding distinct words and their counts");
        String[] words = input.split(" ");
        Map<String, Integer> distinctWords = new HashMap<>();
        for(String w: words) {
            distinctWords.merge(w, 1, (k, v) -> v + 1);
        }
        return distinctWords;
    }
}
```

this task counts the number of distant words in the file content and stores that information in a map.

this is the terminal task for our pipeline, and it outputs the desired result

***Figure 19.11**: Task: Count distinct words*

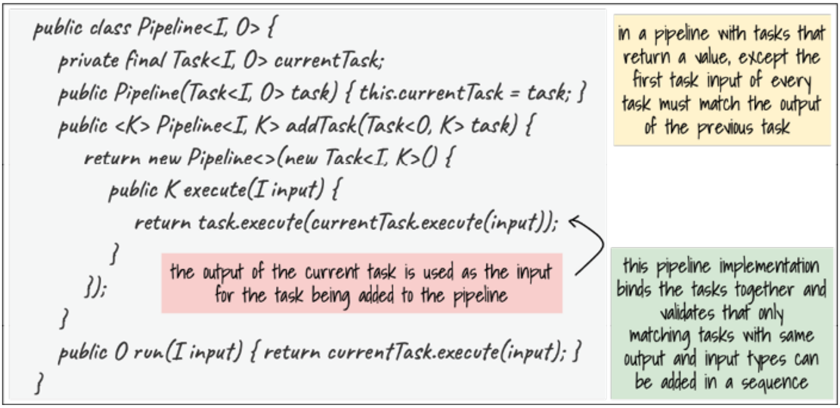
The pipeline design pattern ensures that the tasks run in a pre-decided order, one after another. The pipeline is responsible for processing the entire operation, and each task supports one part of that operation. The pipeline, therefore, must take care of any exceptions or errors that may arise during processing.

The order of the tasks that we discussed earlier was determined while creating the pipeline, and it cannot be altered during its execution.

Constructing a pipeline

The pipeline is the most important part of this pattern. It determines the order in which the tasks must run to complete an operation. The pipeline is mutable in nature and can be expanded or shortened by adding or removing tasks from it. This property of the pipeline allows for easy modification, and the modular nature of it keeps the tasks decoupled from each other.

Figure 19.12 exhibits a pipeline implementation that has two primary functions: add tasks to it and run those tasks to complete the operation:



```
public class Pipeline<I, O> {
    private final Task<I, O> currentTask;
    public Pipeline(Task<I, O> task) { this.currentTask = task; }
    public <K> Pipeline<I, K> addTask(Task<O, K> task) {
        return new Pipeline<>(new Task<I, K>() {
            public K execute(I input) {
                return task.execute(currentTask.execute(input));
            }
        });
    }
    public O run(I input) { return currentTask.execute(input); }
}
```

in a pipeline with tasks that return a value, except the first task input of every task must match the output of the previous task

the output of the current task is used as the input for the task being added to the pipeline

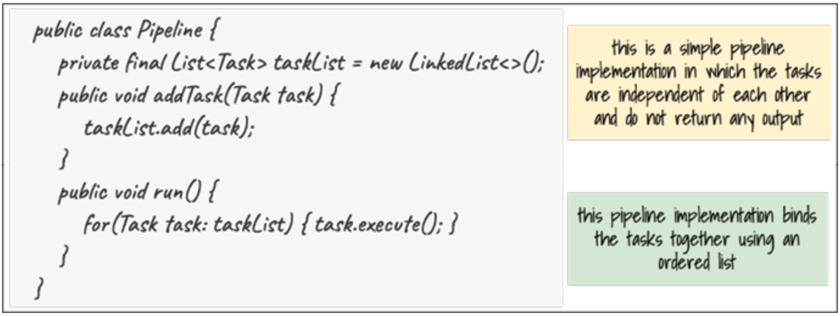
this pipeline implementation binds the tasks together and validates that only matching tasks with same output and input types can be added in a sequence

***Figure 19.12**: Pipeline implementation I*

A pipeline can be implemented in two different ways, depending on whether the tasks produce an output or not. The preceding implementation allows passing the output from one task to another and validates that the output and the input type match for subsequent tasks. It is therefore the preferred implementation choice when tasks are required to communicate with each other while being decoupled.

The preceding implementation clubs the tasks together in a stack-like fashion, wherein each new task addition returns a new pipeline that calls the execute method on the previous task before executing itself. The generic nature of the parameters allows the use of any type of object for communicating between the two adjacent tasks.

A simpler approach to constructing a pipeline is exhibited in *Figure 19.13*. When the tasks do not require communication and passing on information, this implementation should be used instead.



```
public class Pipeline {
    private final List<Task> taskList = new LinkedList<>();
    public void addTask(Task task) {
        taskList.add(task);
    }
    public void run() {
        for(Task task: taskList) { task.execute(); }
    }
}
```

this is a simple pipeline implementation in which the tasks are independent of each other and do not return any output

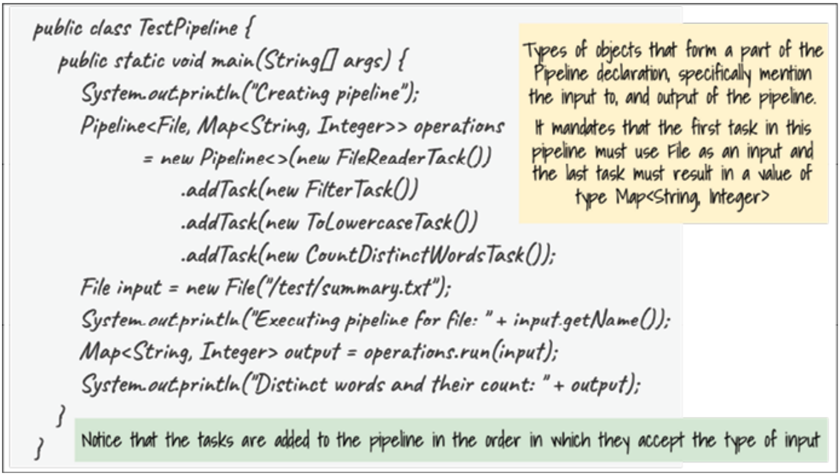
this pipeline implementation binds the tasks together using an ordered list

***Figure 19.13**: Pipeline implementation II*

The preceding implementation clubs the tasks together using a simple list that stores and triggers the tasks in insertion order. In our example, we will use the implementation from *Figure 19.12*.

Testing the pipeline

We have written a test class to verify the working of the pipeline design pattern. Let us execute the same as shown in *Figure 19.14*:



```
public class TestPipeline {
    public static void main(String[] args) {
        System.out.println("Creating pipeline");
        Pipeline<File, Map<String, Integer>> operations
                = new Pipeline<>(new FileReaderTask())
                    .addTask(new FilterTask())
                    .addTask(new ToLowercaseTask())
                    .addTask(new CountDistinctWordsTask());
        File input = new File("/test/summary.txt");
        System.out.println("Executing pipeline for file: " + input.getName());
        Map<String, Integer> output = operations.run(input);
        System.out.println("Distinct words and their count: " + output);
    }
}
```

Types of objects that form a part of the Pipeline declaration, specifically mention the input to, and output of the pipeline. It mandates that the first task in this pipeline must use File as an input and the last task must result in a value of type Map<String, Integer>

Notice that the tasks are added to the pipeline in the order in which they accept the type of input

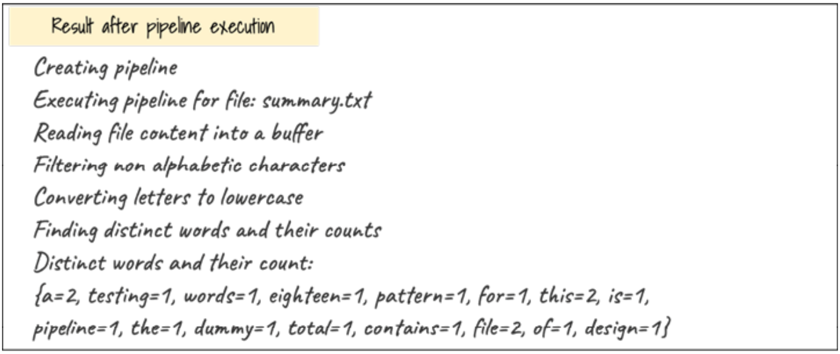
***Figure 19.14**: Testing the pipeline*

In the preceding example, a new pipeline is created with four different tasks that are added to it in a sequential order.

An object of type **file**, which is the input type expected by the first task, was created and passed to the pipeline, and the rest was managed by the pipeline itself. Notice that no other input types are required, as each task is producing and consuming some object in an automated fashion.

The two generic parameter types mentioned during the pipeline creation refer to the input type and the output type of the pipeline.

Figure 19.15 exhibits the output of triggering the preceding pipeline:



```
Result after pipeline execution
Creating pipeline
Executing pipeline for file: summary.txt
Reading file content into a buffer
Filtering non alphabetic characters
Converting letters to lowercase
Finding distinct words and their counts
Distinct words and their count:
{a=2, testing=1, words=1, eighteen=1, pattern=1, for=1, this=2, is=1,
pipeline=1, the=1, dummy=1, total=1, contains=1, file=2, of=1, design=1}
```

***Figure 19.15**: Test result*

Summary

The pipeline design pattern aims to execute an operation by splitting it into multiple subtasks that are represented by their own classes and are decoupled from each other. It utilizes abstraction and promotes modular development to allow tasks to be developed and run as separate modules. The pipeline itself is extensible and allows adding tasks to it at any step, be it the first, the last, or between any two existing tasks.

Individual tasks can be reused in different pipelines and added at different positions to exhibit different behaviors. The pipeline can be implemented in two different ways based on whether the tasks produce and consume data in the form of objects.

In *Figure 19.16*, the pipeline design pattern is reimaged with the help of the classes that we have used in our example:

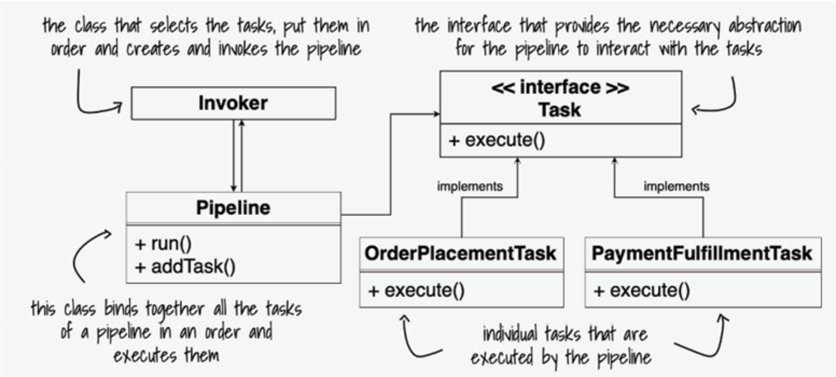

the class that selects the tasks, put them in order and creates and invokes the pipeline
the interface that provides the necessary abstraction for the pipeline to interact with the tasks
Invoker
<< interface >>
Task
+ execute()
implements
implements
Pipeline
+ run()
+ addTask()
OrderPlacementTask
+ execute()
PaymentFulfillmentTask
+ execute()
this class binds together all the tasks of a pipeline in an order and executes them
individual tasks that are executed by the pipeline

***Figure 19.16**: Pipeline design pattern—Defined*

Advantages and drawbacks

The following list outlines the benefits of pipeline design pattern:

- By splitting a complex task into subtasks, the system becomes easier to maintain.
- Modular development is promoted through the use of our own classes to represent tasks.
- The promotion of reusability is achieved by allowing tasks to be used in various pipelines.
- It promotes extensibility and allows for altering the behavior of a pipeline by adding new tasks without modifying the existing ones.

The drawbacks of the pipeline design pattern are:

- Managing too many tasks in a pipeline can increase the overall complexity.
- Multiple context switches and data transfer between tasks could have a slight impact on performance when using a pipeline.

Usage

The following list outlines some of the scenarios that are relevant to the pipeline design pattern:

- It can be used to reduce the complexity of a task by splitting it into subtasks, making it more manageable.
- When there is a need for a slight change in behavior while keeping the structure, a pipeline can be utilized as a template.

Conclusion

Pipeline design pattern allows to breakdown a large complex task into smaller, manageable ones. It supports the principle of separation of concerns. It promotes extensibility and reusability with the help of classes that encapsulate the functionality of an entire task. The tasks can be easily replaced to alter the behavior of a pipeline without impacting other tasks.

In the next chapter, we will learn about the various design principles and how they can be utilized to simplify the process of software development.

Questions

In this section, we will sharpen our brains by indulging in some fun brain activities. Let us get started.

Find the odd one out

1. Pipeline design pattern:
    a. Promotes separation of concern
    b. Is a behavioral design pattern
    c. Promotes reusability
    d. Increases complexity
2. Tasks in a pipeline:
    a. Are decoupled from each other
    b. Can be reused
    c. Can be modified during pipeline execution
    d. Are only relevant for a specific pipeline

Select two correct answers

1. A pipeline:
    a. Can have only one task
    b. Supports the principle of separation of concerns
    c. Restricts reusability
    d. Increases coupling
2. Tasks are:
    a. Reusable
    b. Immutable

c. Interchangeable
d. Tightly coupled

Answers

This section provides the answers to the questions given.

Find the odd one out

1. Increases complexity
2. Are only relevant for a specific pipeline

Select two correct answers

1. Can have only one task, supports the principle of separation of concerns
2. Reusable, interchangeable

CHAPTER 20

Beyond Design Patterns

Introduction

In the previous chapters, we discussed multiple design patterns, mastered their implementation, and learned about their advantages and disadvantages. In this chapter, we will follow up and expand our understanding of a good system design by exploring some of the design principles that are derived from the previous architecture and design experiences, which form the basis of many of the design patterns. We will explore some of the anti-patterns that must be avoided when finalizing a system design, as they are usually ineffective and risky of being counterproductive.

To conclude our journey towards improving the system design and providing effective solutions, we would like to reiterate that the design patterns that we explored in this book are not the only ones and that there are multiple others that are equally important to be explored and learned.

Structure

This chapter covers the following topics:

- Quick recap
- Summary of patterns
- Design principles
- Anti-patterns

Objectives

The objective of this chapter is to summarize the knowledge the readers gathered in the past chapters, understand some of the design principles that are an integral part of software design, and learn about some of the anti-patterns that should be avoided while designing software applications.

Quick recap

We learned about three different categories of software design patterns, which are creational, structural, and behavioral. We also explored some of the design patterns that fall under each of these categories and learned to implement them with close to actual application examples. The following list outlines a quick recap:

- **Creational**: Provides a mechanism for object creation in an efficient and controlled manner while improving code reusability. Promotes the idea of hiding object composition from the outside world.
- **Structural**: Relates to the idea of forming larger structures by assembling various objects or their classes together and provides new functionalities. They do not modify existing classes or objects, but instead add new features on top of them.
- **Behavioral**: Defines the relationship between objects and how they interact with each other. Aims to reduce the communication complexity by using abstraction to decouple the classes and promote interaction with interfaces.

  Figure 20.1 demonstrates the design patterns that we explored in the previous chapters, along with the categories they belong to:

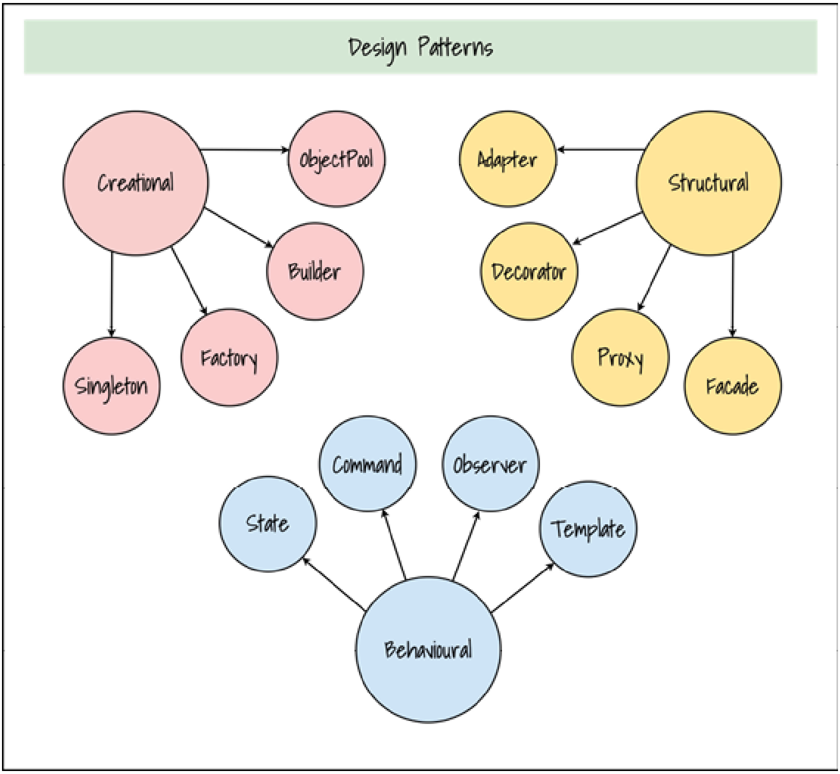
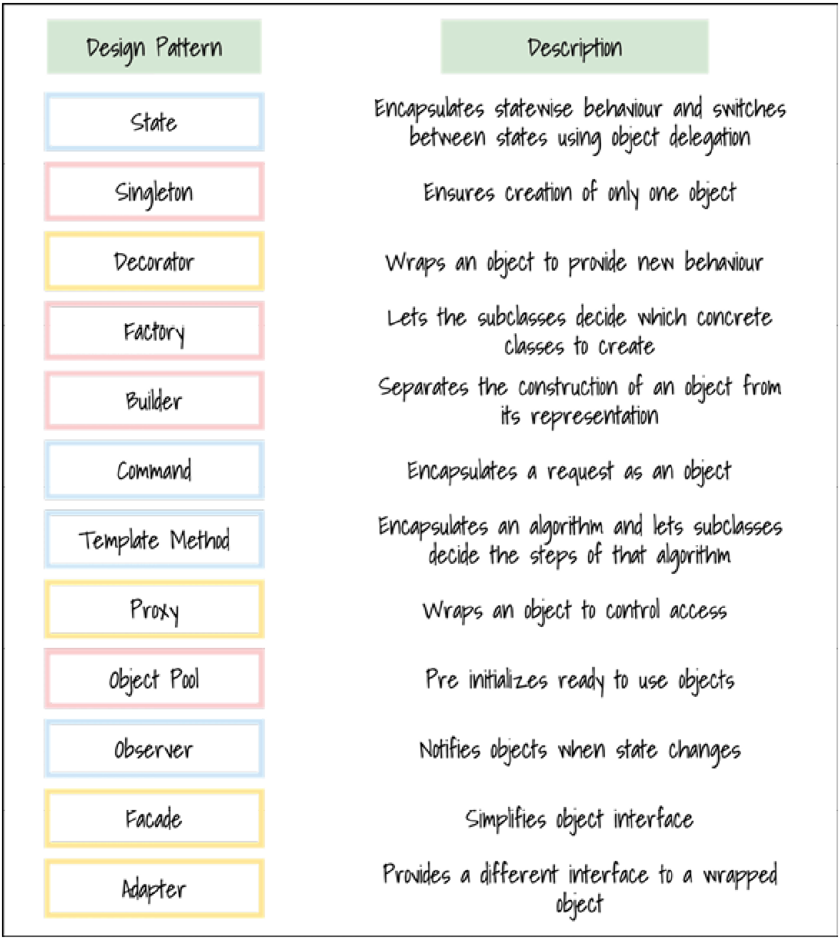

Design Patterns
Creational
ObjectPool
Builder
Factory
Singleton
Adapter
Structural
Decorator
Proxy
Facade
Command
Observer
State
Template
Behavioural

***Figure 20.1**: Design patterns*

The design patterns serve the objective of streamlining and simplifying the architecture of a system. They make use of the four pillars of object-oriented programming, that is, abstraction, encapsulation, polymorphism, and inheritance, and provide solutions for recurring and general software design problems.

Summary of patterns

Here is a quick and precise summary of the design patterns we explored in this book. We have listed down the design patterns and the objectives they serve. *Figure 20.2* describes the design patterns we learned throughout the book, in simple one-liners:



Design Pattern	Description
State	Encapsulates statewise behaviour and switches between states using object delegation
Singleton	Ensures creation of only one object
Decorator	Wraps an object to provide new behaviour
Factory	Lets the subclasses decide which concrete classes to create
Builder	Separates the construction of an object from its representation
Command	Encapsulates a request as an object
Template Method	Encapsulates an algorithm and lets subclasses decide the steps of that algorithm
Proxy	Wraps an object to control access
Object Pool	Pre initializes ready to use objects
Observer	Notifies objects when state changes
Facade	Simplifies object interface
Adapter	Provides a different interface to a wrapped object

***Figure 20.2**: Summary of patterns*

It points out the intent and usefulness of these design patterns in system design that is both imperative and purposeful.

Classification of class versus object patterns

The design patterns can be further classified by another attribute—whether the pattern deals with classes or objects. The classification is best known as class patterns and object patterns.

Class patterns describe the relationship between classes via inheritance. These relationships are established at compile time.

Object patterns describe the relationship between objects, primarily using composition. These relationships are established at runtime and are more flexible and dynamic compared to those established using the class patterns.

Figure 20.3 demonstrates the classification of various design patterns based on their interaction with either classes or objects:

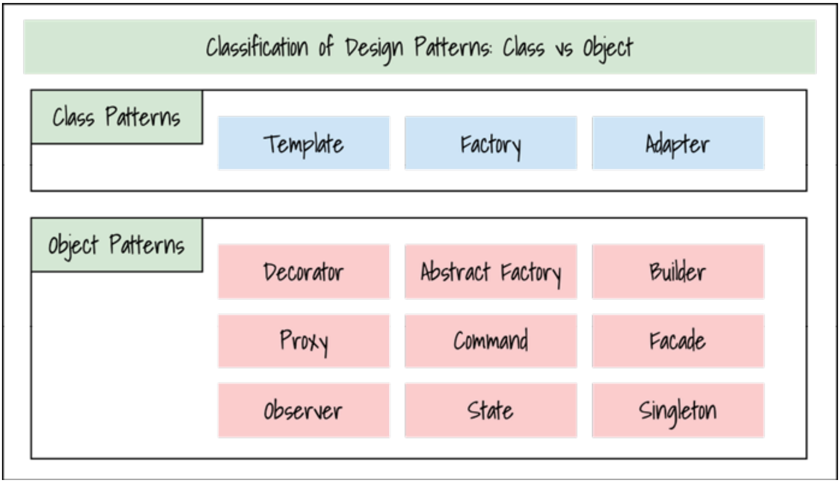

Classification of Design Patterns: Class vs Object
Class Patterns
Template
Factory
Adapter
Object Patterns
Decorator
Abstract Factory
Builder
Proxy
Command
Facade
Observer
State
Singleton

***Figure 20.3**: Classification of design—Class versus object*

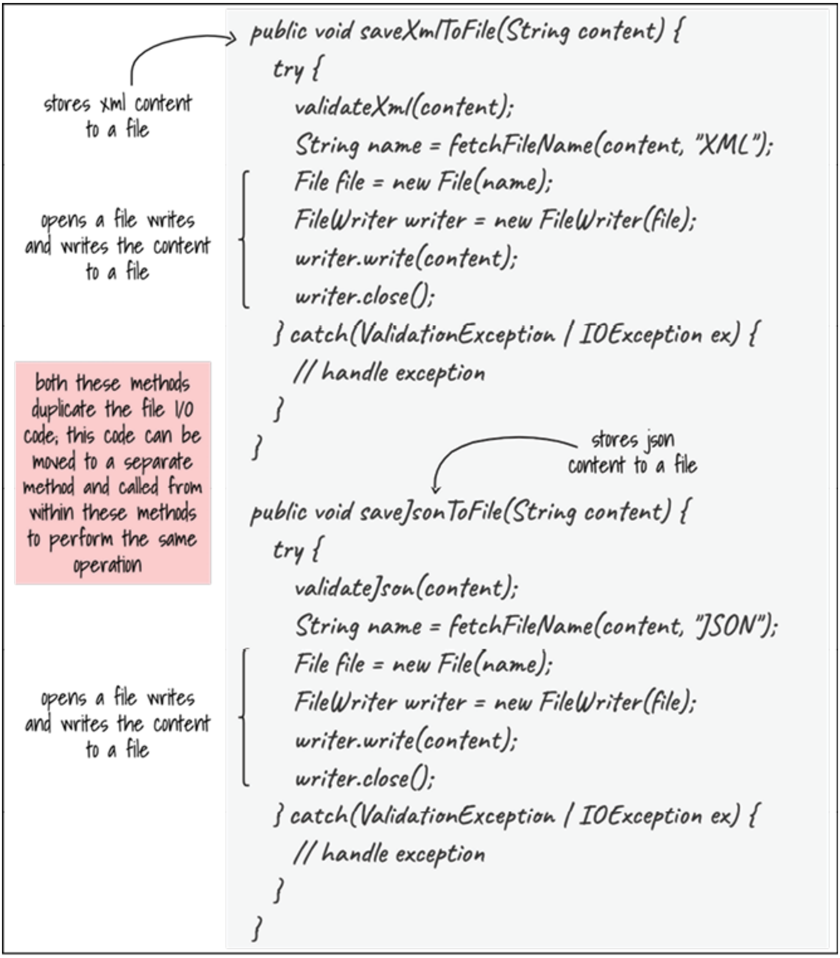
Design principles

Design principles are the guidelines or recommendations that streamline the design and reduce the complexity of an application. These principles are devised following years of software development best practices. They bring a kind of maturity to software development and show us how the code should be written to avoid a bad design. In this section, we will explore some of these design principles and learn about the benefits that come with them.

Don't Repeat Yourself

Famously known as the DRY principle, this principle asserts that the same code should not be repeated in multiple places. Instead, the common logic should be extracted into methods.

Figure 20.4 demonstrates two methods that contain duplicate code of writing to a file:



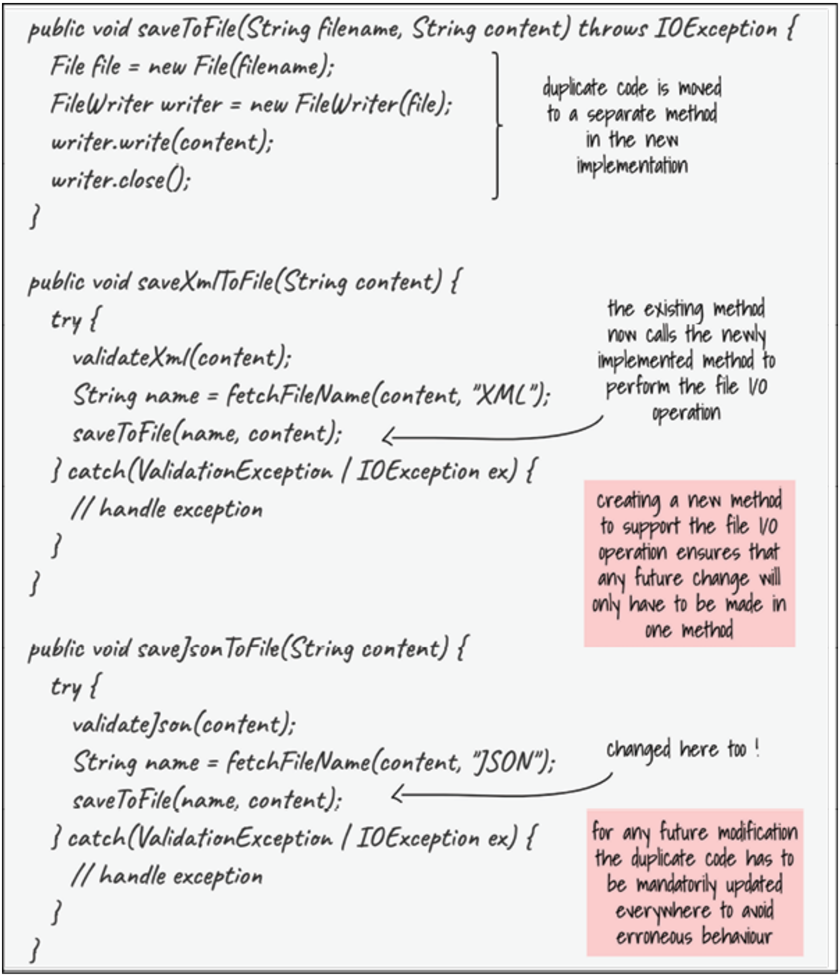
stores xml content to a file
opens a file writes and writes the content to a file
stores json content to a file
opens a file writes and writes the content to a file

both these methods duplicate the file I/O code; this code can be moved to a separate method and called from within these methods to perform the same operation

```
public void saveXmlToFile(String content) {
  try {
    validateXml(content);
    String name = fetchFileName(content, "XML");
    File file = new File(name);
    FileWriter writer = new FileWriter(file);
    writer.write(content);
    writer.close();
  } catch(ValidationException | IOException ex) {
    // handle exception
  }
}

public void saveJsonToFile(String content) {
  try {
    validateJson(content);
    String name = fetchFileName(content, "JSON");
    File file = new File(name);
    FileWriter writer = new FileWriter(file);
    writer.write(content);
    writer.close();
  } catch(ValidationException | IOException ex) {
    // handle exception
  }
}
```

***Figure 20.4**: Design principle—DRY I*

Figure 20.5 demonstrates the application of the DRY principle by extracting the common code into a separate method:



```
public void saveToFile(String filename, String content) throws IOException {
  File file = new File(filename);
  FileWriter writer = new FileWriter(file);
  writer.write(content);
  writer.close();
}

public void saveXmlToFile(String content) {
  try {
    validateXml(content);
    String name = fetchFileName(content, "XML");
    saveToFile(name, content);
  } catch(ValidationException | IOException ex) {
    // handle exception
  }
}

public void saveJsonToFile(String content) {
  try {
    validateJson(content);
    String name = fetchFileName(content, "JSON");
    saveToFile(name, content);
  } catch(ValidationException | IOException ex) {
    // handle exception
  }
}
```

duplicate code is moved to a separate method in the new implementation

the existing method now calls the newly implemented method to perform the file I/O operation

creating a new method to support the file I/O operation ensures that any future change will only have to be made in one method

changed here too !

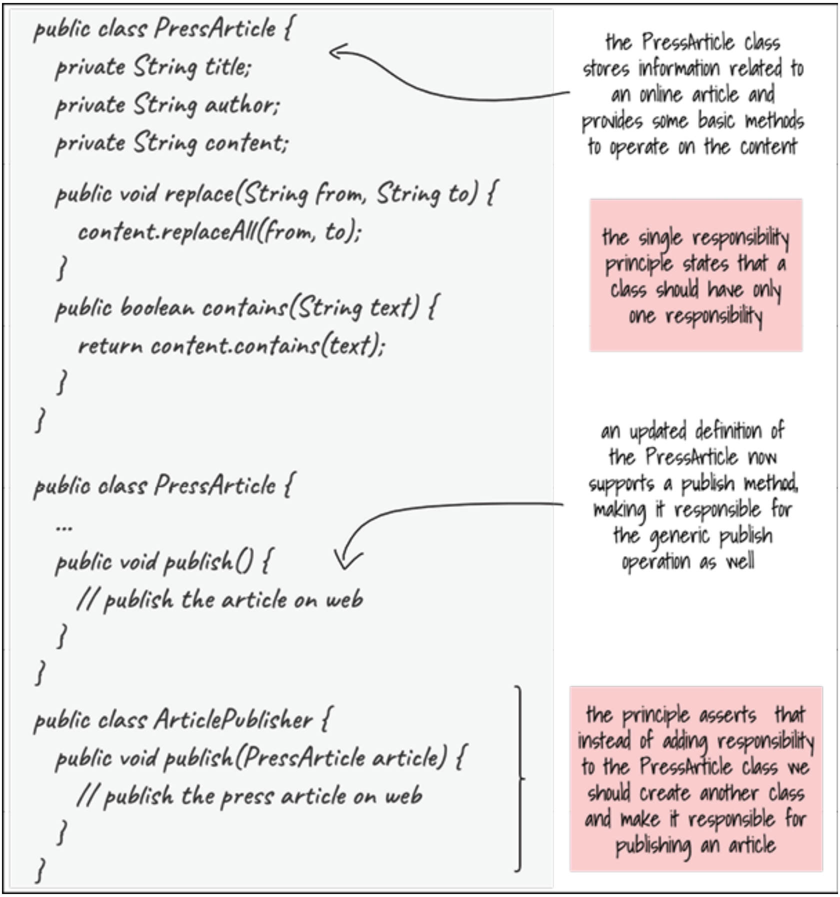
for any future modification the duplicate code has to be mandatorily updated everywhere to avoid erroneous behaviour

***Figure 20.5**: Design principle—DRY II*

An example of a common logic is reading a file or using the same literal in multiple places. A method could be constructed for reading a file, and the literal could be stored in a final variable. Extracting the common logic into methods or the use of final variables to store repeatedly used values reduces the number of places where a change would have to be replicated if the same code is used at various places. It also reduces the number of tests that need to be performed for testing the same code at multiple places. This principle promotes code reusability, reduces testing effort, and lowers the risk of introducing changes to the system.

Single responsibility

The *single responsibility principle* asserts that each construct should only perform one single task; a construct could either be a class or a method. In other words, there should only be one reason for the constructs to change. *Figure 20.6* demonstrates the application of the single responsibility principle:



```
public class PressArticle {
  private String title;
  private String author;
  private String content;

  public void replace(String from, String to) {
    content.replaceAll(from, to);
  }
  public boolean contains(String text) {
    return content.contains(text);
  }
}
```

the PressArticle class stores information related to an online article and provides some basic methods to operate on the content

the single responsibility principle states that a class should have only one responsibility

```
public class PressArticle {
  ...
  public void publish() {
    // publish the article on web
  }
}
```

an updated definition of the PressArticle now supports a publish method, making it responsible for the generic publish operation as well

```
public class ArticlePublisher {
  public void publish(PressArticle article) {
    // publish the press article on web
  }
}
```

the principle asserts that instead of adding responsibility to the PressArticle class we should create another class and make it responsible for publishing an article

***Figure 20.6**: Design principle—Single responsibility*

It does so by creating another class for publishing press articles instead of the existing class.

This principle states that if the construct is responsible for more than one functionality, then it should be split into as many constructs as the functionalities themselves. It helps to regulate the effect of change on a class. By ensuring that a class performs only one task, it limits the scope of the change and the domino effect it could have if the same class served multiple purposes. In short, by clearly defining the responsibility that a class has, a change in that class is only required when there is a change in how that responsibility is performed. It promotes the development of small, simple classes that have fewer dependencies. It reduces the effort of testing by reducing the complexity of classes.

Dependency inversion

This depends upon abstractions, not on concrete classes. The *dependency inversion* principle sounds very similar to the *favor interface over implementation* principle, but there is a slight difference. This principle asserts that the high-level classes should be decoupled from the low-level classes by introducing an abstraction layer. It also inverts the dependency. With this change, the details or the concrete implementations are written following the abstraction instead of the other way around.

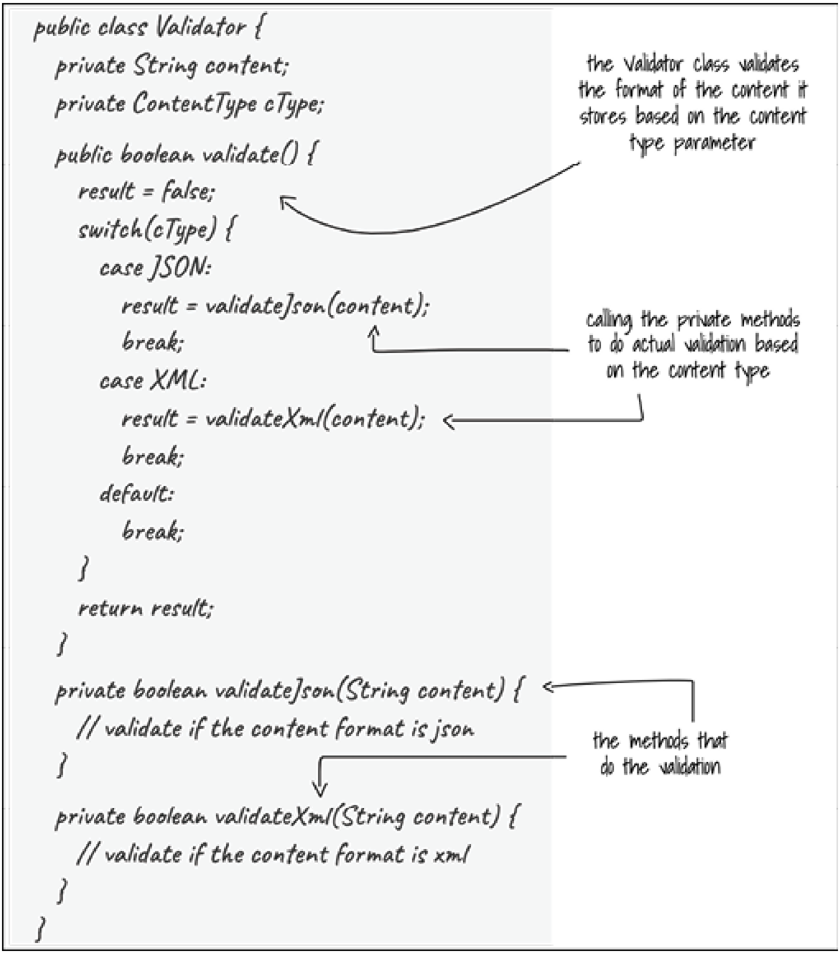
By decoupling the two layers, they are no longer dependent and can be modified in isolation without affecting each other. Due to the abstraction, it becomes straightforward to replace an existing implementation with another, without the other side of the abstraction being aware of it.

It promotes interaction with interfaces and reduces coupling. The classes are no longer dependent and can be developed in isolation.

Encapsulate what varies

Encapsulation is the process of binding the state and behavior of a class together. It hides the internal complexity of a class and prevents it from being exposed. By using access modifiers and methods, an object’s state can be guarded from direct access.

Figure 20.7 demonstrates a class that is open for changes and will remain so in the future as well. Any addition to this class requires retesting the existing behavior as well:



```
public class Validator {
  private String content;
  private ContentType cType;

  public boolean validate() {
    result = false;
    switch(cType) {
      case JSON:
        result = validateJson(content);
        break;
      case XML:
        result = validateXml(content);
        break;
      default:
        break;
    }
    return result;
  }
  private boolean validateJson(String content) {
    // validate if the content format is json
  }
  private boolean validateXml(String content) {
    // validate if the content format is xml
  }
}
```

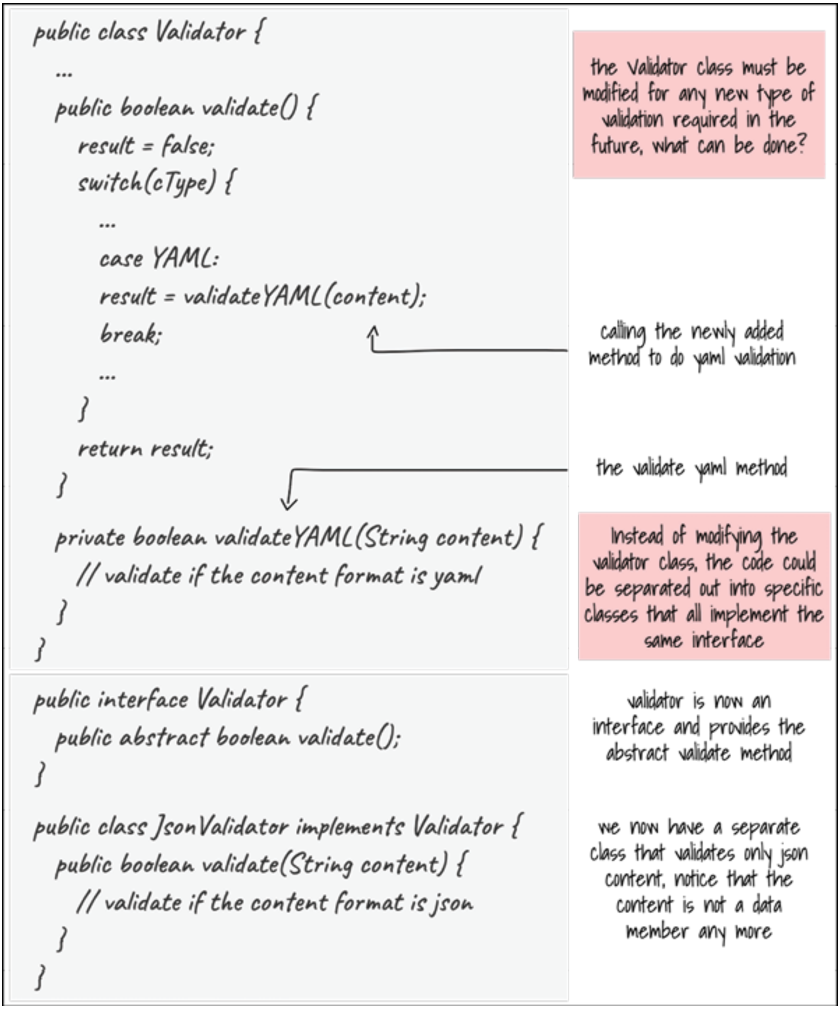
the Validator class validates the format of the content it stores based on the content type parameter

calling the private methods to do actual validation based on the content type

the methods that do the validation

***Figure 20.7**: Design principle—Encapsulate what varies I*

The principle, *encapsulate what varies*, asserts that those aspects of an application that frequently change should be separated from what stays the same. This way, the change can be isolated and tested separately without affecting the unchanged behavior. *Figure 20.8* demonstrates the application of the *encapsulate what varies* principle:



```
public class Validator {
  ...
  public boolean validate() {
    result = false;
    switch(cType) {
      ...
      case YAML:
      result = validateYAML(content);
      break;
      ...
    }
    return result;
  }
  private boolean validateYAML(String content) {
    // validate if the content format is yaml
  }
}
```

the Validator class must be modified for any new type of validation required in the future, what can be done?

calling the newly added method to do yaml validation

the validate yaml method

Instead of modifying the validator class, the code could be separated out into specific classes that all implement the same interface

```
public interface Validator {
  public abstract boolean validate();
}

public class JsonValidator implements Validator {
  public boolean validate(String content) {
    // validate if the content format is json
  }
}
```

validator is now an interface and provides the abstract validate method

we now have a separate class that validates only json content, notice that the content is not a data member any more

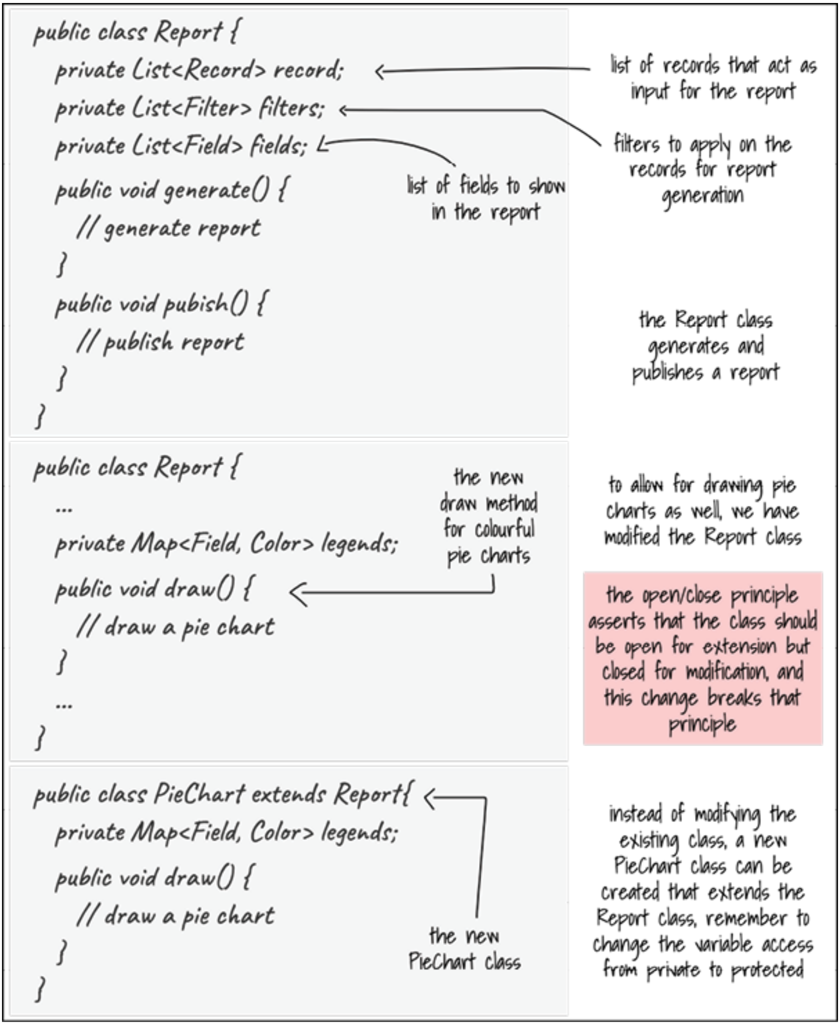
Figure 20.8: *Design principle—Encapsulate what varies II*

The principle mentions creating an interface and its concrete implementations to support different forms of validation, an improvement over the previous implementation.

Open closed

The *open close* principle asserts that the classes should be open for extension but closed for modification. It advocates the use of inheritance to add additional behavior to a class, instead of modifying it directly.

Figure 20.9 demonstrates the principle by extending the **Report** class to provide additional features, instead of modifying it:



```
public class Report {
  private List<Record> record;
  private List<Filter> filters;
  private List<Field> fields;
  public void generate() {
    // generate report
  }
  public void pubish() {
    // publish report
  }
}
```

list of records that act as input for the report

filters to apply on the records for report generation

list of fields to show in the report

the Report class generates and publishes a report

```
public class Report {
  ...
  private Map<Field, Color> legends;
  public void draw() {
    // draw a pie chart
  }
  ...
}
```

the new draw method for colourful pie charts

to allow for drawing pie charts as well, we have modified the Report class

the open/close principle asserts that the class should be open for extension but closed for modification, and this change breaks that principle

```
public class PieChart extends Report{
  private Map<Field, Color> legends;
  public void draw() {
    // draw a pie chart
  }
}
```

the new PieChart class

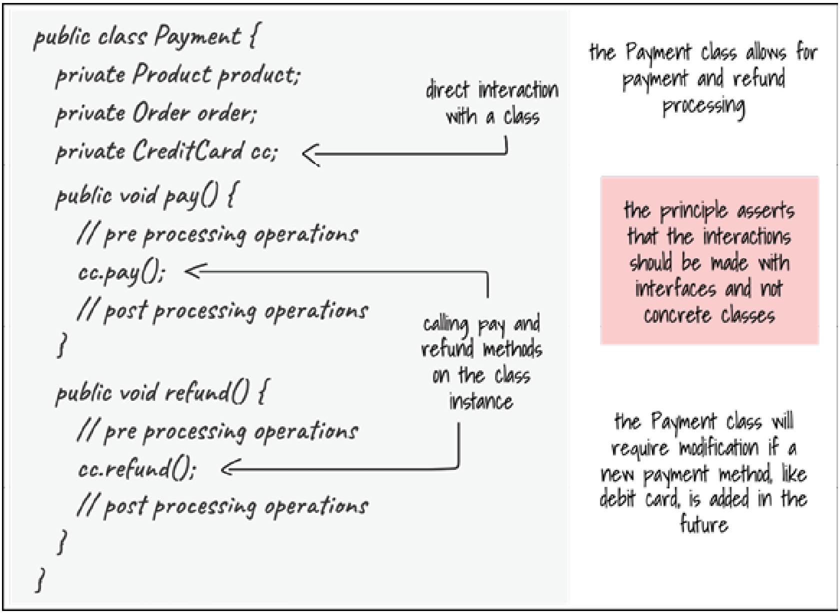
instead of modifying the existing class, a new PieChart class can be created that extends the Report class, remember to change the variable access from private to protected

Figure 20.9*: Design principle: Open-closed*

This principle promotes modifying an existing class to ensure better future maintenance, limits the testing effort to the newly added class or method, and reduces risk.

Favor interface over implementation

More commonly known as **program to an interface, not an implementation**, this design principle asserts that interaction with interfaces should be chosen over interaction with concrete implementations. What this means is that whenever a class type interacts with or is used within another class type, the reference type should be an interface. *Figure 20.10* demonstrates a class that maintains a has-a relationship with concrete classes instead of interfaces:



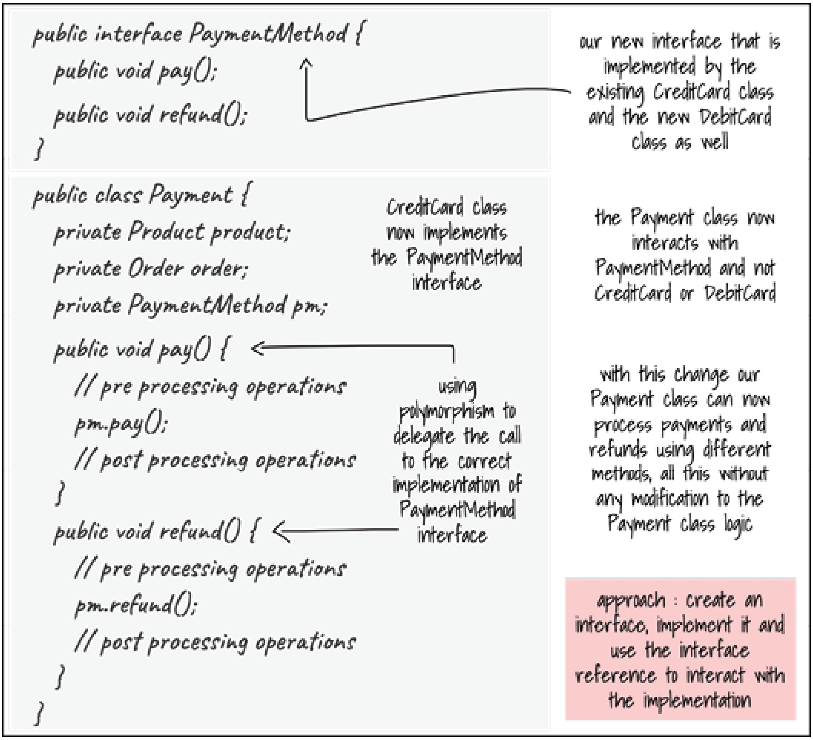
```
public class Payment {
   private Product product;
   private Order order;
   private CreditCard cc;
   public void pay() {
      // pre processing operations
      cc.pay();
      // post processing operations
   }
   public void refund() {
      // pre processing operations
      cc.refund();
      // post processing operations
   }
}
```

direct interaction with a class
calling pay and refund methods on the class instance
the Payment class allows for payment and refund processing
the principle asserts that the interactions should be made with interfaces and not concrete classes
the Payment class will require modification if a new payment method, like debit card, is added in the future

***Figure 20.10**: Design principle—Favor interface over implementation*

In the preceding figure, **CreditCard** is a class and not an interface. This class is a potential candidate for behavior change if a new payment method is added to it in the future.

Figure 20.11 demonstrates interaction with interfaces instead of concrete classes. This one-time change to the existing implementation of the **Payment** class will allow it to be used with many different payment methods, old and new alike, without any additional modification:



```
public interface PaymentMethod {
   public void pay();
   public void refund();
}
```

```
public class Payment {
   private Product product;
   private Order order;
   private PaymentMethod pm;
   public void pay() {
      // pre processing operations
      pm.pay();
      // post processing operations
   }
   public void refund() {
      // pre processing operations
      pm.refund();
      // post processing operations
   }
}
```

our new interface that is implemented by the existing CreditCard class and the new DebitCard class as well

CreditCard class now implements the PaymentMethod interface

the Payment class now interacts with PaymentMethod and not CreditCard or DebitCard

using polymorphism to delegate the call to the correct implementation of PaymentMethod interface

with this change our Payment class can now process payments and refunds using different methods, all this without any modification to the Payment class logic

approach : create an interface, implement it and use the interface reference to interact with the implementation

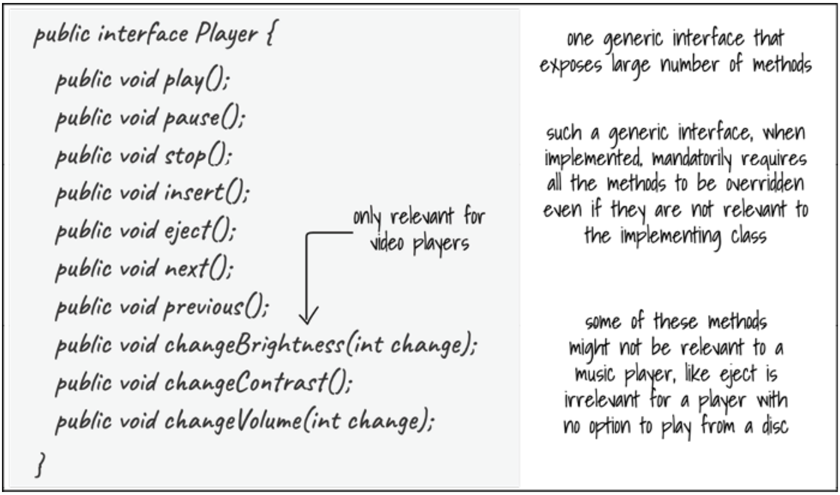
Figure 20.11*: Design principle—Favor interface over implementation*

This principle is very useful for building solutions that support dynamic behavior without modifying the existing code. Programming to interfaces leads to a flexible, reusable, and incremental approach to software development, where new behaviors can be added to the existing system by adding new implementations to the existing code.

Interface segregation

The *interface segregation* principle asserts that the specialized interfaces should be favored over generic ones. An interface that exposes many behaviors is hard to manage and leads to unnecessary implementations. A lean interface that provides specialized behavior is both easy to implement and manage.

Figure 20.12 demonstrates a generic interface for media players:



```
public interface Player {
    public void play();
    public void pause();
    public void stop();
    public void insert();
    public void eject();
    public void next();
    public void previous();
    public void changeBrightness(int change);
    public void changeContrast();
    public void changeVolume(int change);
}
```

only relevant for video players

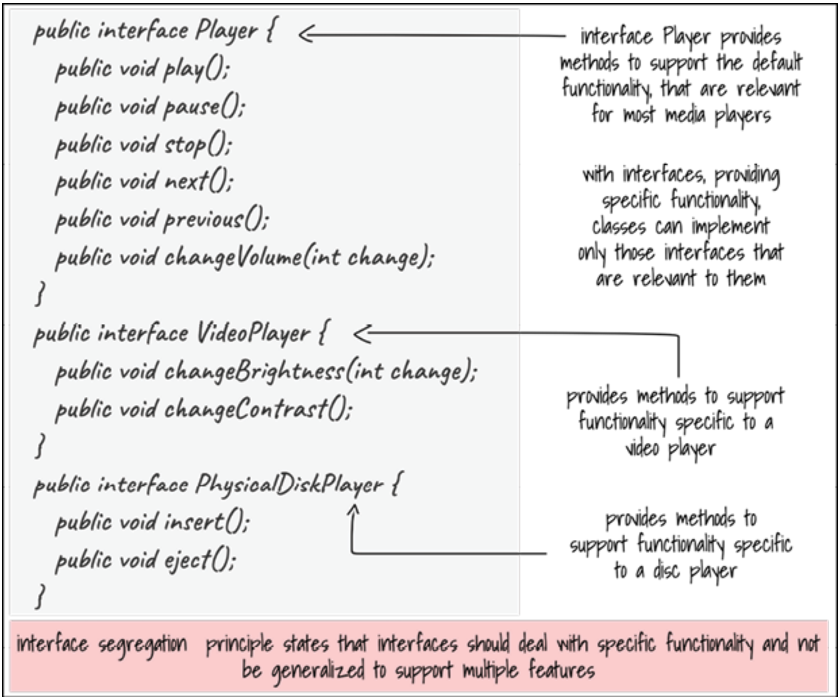
one generic interface that exposes large number of methods

such a generic interface, when implemented, mandatorily requires all the methods to be overridden even if they are not relevant to the implementing class

some of these methods might not be relevant to a music player, like eject is irrelevant for a player with no option to play from a disc

***Figure 20.12**: Design principle—Interface segregation I*

Figure 20.13 demonstrates how the generic **Player** interface can be broken into multiple smaller interfaces to support specialized behaviors:



```
public interface Player {
    public void play();
    public void pause();
    public void stop();
    public void next();
    public void previous();
    public void changeVolume(int change);
}
public interface VideoPlayer {
    public void changeBrightness(int change);
    public void changeContrast();
}
public interface PhysicalDiskPlayer {
    public void insert();
    public void eject();
}
```

interface Player provides methods to support the default functionality, that are relevant for most media players

with interfaces, providing specific functionality, classes can implement only those interfaces that are relevant to them

provides methods to support functionality specific to a video player

provides methods to support functionality specific to a disc player

interface segregation principle states that interfaces should deal with specific functionality and not be generalized to support multiple features

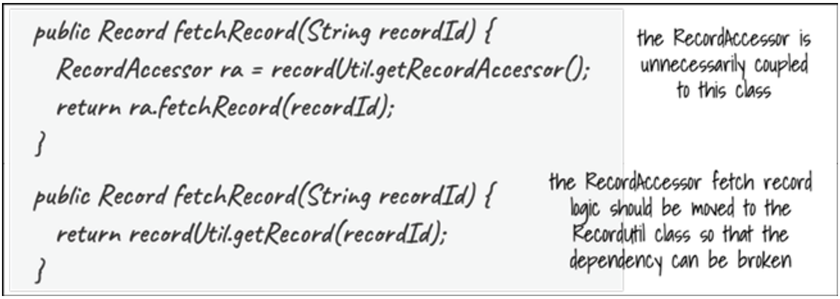
***Figure 20.13**: Design principle—Interface segregation II*

An interface with multiple behaviors is often termed as polluted. Such an interface should be broken down into multiple specialized interfaces, each responsible for behaviors that are highly cohesive and taken from the generic interface. A class can implement multiple specialized interfaces to expose relevant behavior instead of implementing a larger interface and overriding all the methods.

Least knowledge

The principle of *least knowledge* asserts that the classes should not be highly coupled to each other. It guides us to reduce the interaction between classes to just a few necessary ones.

Figure 20.14 demonstrates the use of the principle of *least knowledge* to reduce coupling and dependency among class implementations:



```
public Record fetchRecord(String recordId) {
  RecordAccessor ra = recordUtil.getRecordAccessor();
  return ra.fetchRecord(recordId);
}
```

the RecordAccessor is unnecessarily coupled to this class

```
public Record fetchRecord(String recordId) {
  return recordUtil.getRecord(recordId);
}
```

the RecordAccessor fetch record logic should be moved to the Recordutil class so that the dependency can be broken

***Figure 20.14**: Design principle—Least knowledge*

It prevents us from designing highly coupled class structures and saves us from a cascading effect triggered by a change in a class that is coupled with multiple other classes. A single change could trigger a batch of changes in all the related classes, and those changes can lead to more changes.

The principle states that we should only invoke methods that belong to:

- The object itself.
- An object that is referenced by this object.
- An object that is passed as an argument to a method.
- An object that the method creates or instantiates.

Building lots of dependencies leads to high maintenance costs and a weak, delicate architecture that can be broken because of its own internals.

There are many more design principles that are highly effective and can be used to improve everyday design solutions. The design principles are so exhaustive that they deserve a book of their own, not just a chapter.

Anti-patterns

An anti-pattern is a formulated response to solve a common, recurring problem, but is often ineffective and counterproductive. While the design patterns are generally considered a good development practice, the anti-patterns are undesirable and should be avoided. Anti-patterns are the opposite of design patterns and are often mistaken for good design practices.

Initially, an anti-pattern may seem to be the right approach for problem-solving, but they often lead to trouble in the long run. They are often considered bad design that leads to messy code and faulty architecture. There are almost as many, or maybe more, anti-patterns as there are design patterns, and they deserve a book of their own. In this chapter, we will only discuss a few of them that are often used in regular practice.

Interface for constants

Using interfaces just to define constants and having classes implement them is one practice that is commonly followed to gain access to constant values. Such an interface not only deviates from its core objective, that is, to define the actions that an implementing class can perform, but it also exposes the implementation details in the form of those constants.

It could be argued that keeping constants in an interface increases the convenience with which they can be accessed, but they also end up being publicly exposed to all the classes that implement the interface and probably do not need them.

Constants should ideally be kept inside:

- The class definition if they are only relevant to that class.

- The parent class definition is if it has multiple child classes, where the constant is relevant.
- The interface of the constants is relevant to all or most of its implementations.
- Another class if the constants are relevant to a wide range of classes, not necessarily in the same hierarchy.

One should try to avoid using interfaces for keeping a constant value because it could be difficult to remove them later for the sake of backward compatibility.

God object

An object that provides multiple functionalities and handles too many responsibilities is often termed a **god object**.

At times, we end up writing so much logic in a single object that it becomes a know-all, do-all construct. Such an object is in sharp contrast with the single responsibility principle and is usually related to multiple functionalities. While the responsibilities should ideally be divided among multiple objects that interact with each other to solve a computational or business scenario, one large, heavily dependent object ends up doing everything.

Maintaining such an object is not only difficult, but a nightmare at the least when it comes to testing. Even a small change might end up requiring a rigorous amount of testing. A large object defeats the purpose of having a low-maintenance, replaceable, and scalable architecture.

Classes should be made smaller and less responsible. They should be cohesive and reflect the purpose of their existence.

Caching irregularities

Caching data for future reuse is a common approach to speed up performance and reduce the load on the databases. The caching mechanism is often written as a separate layer that exposes certain **application programming interfaces** (**APIs**) to interact with the clients. While caching has become an integral part of the system design, it is usually let down by incomplete or bad implementations.

Having multiple caches and a few wrong implementations can reverse all the good things that caching provides. The problem is not with the use of multiple caches, but with implementations that write a value to one cache and read it from another. It could end up with multiple unnecessary calls to the backend and could be highly difficult to trace and correct.

Another wrong practice is not updating the cache after a certain database operation. It is highly significant when a deletion takes place, and the cache is not properly updated, resulting in stale or wrong data being shown or sent in response.

Hard code

Embedding a changeable piece of information directly in the implementation is called hard-coding. Do not confuse it with constants; they are more aligned with the implementation. Examples of hard-coding range from links to services, login credentials, and paths to internal resources. Hard-coding mostly requires recompilation and server restarts for the changes to take effect. They cannot be changed at runtime or by any user. They require code changes that can only be addressed by a developer.

Ideally, login credentials should not be stored at all, and links to services should be behind a gateway that the application can connect to. Yet, if there is a need to store such information, it can be stored in files, outside the code base, and read during system initialization or when there is an alteration to that file.

Boat anchor

Obsolete or not-in-use code that is still a part of the code base is called a **boat anchor**. Retaining outdated functionality in the system is an invitation to an invisible risk. Although the unused modules, classes, or any other structs seem highly unlikely to cause a problem, they still can. Here is a list of possible risks that arise due to disused code:

- It could be unknowingly referred to and used in the active code, leading to some unwarranted behavior.
- It could impact the performance of a system by starting up some scheduled or queueing tasks when they are no longer needed.

- It increases the compilation time.
- Any active singletons or spring beans may use valuable system resources.
- Developers may end up debugging the wrong code before they eventually get to know that it is not related to the problem they are solving.

Obsolete code, even if it may be required in the future, should be kept in a separate repository from where it can be referred to or reinstated when required.

Copy and paste

Code redundancy and duplication are what is termed as copy and paste programming. Although it may be useful in some scenarios, like boilerplate code, loop unrolling, or code segments that are short-lived and that do not require maintenance, it is mostly better to refactor and extract the duplicate code into methods or classes.

Extracting redundant code into methods helps with easy maintenance and fast development, as one does not need to reinvent and maintain all copies of the code. Another way to approach the problem is to use existing libraries that are well tested and provide efficient solutions rather than programming the entire code base afresh.

No silver bullets

The simple act of believing that one solution is fit for many, if not all the technical problems is termed as a silver bullet assumption. Every problem is unique, and although there may be valid reasons to use a tested approach to solve many similar problems, it is not always the best one.

There may be other approaches that are more performant and better designed to solve a problem but we choose the one we prefer the most, usually due to a cognitive bias.

Conclusion

In this concluding chapter, we discussed some of the important design principles and anti-patterns. The design principles guide us on how to improve our design, and the anti-patterns warn us about the bad practices.

Questions

In this section, we will sharpen our brains by indulging in some fun brain activities. Let us get started.

Find the odd one out

1. Which one of these is not an anti-pattern?
    a. No silver bullets
    b. Hard code
    c. Least knowledge
    d. God object
2. Which one of these is an anti-pattern?
    a. Boat anchor
    b. Least knowledge
    c. Open closed
    d. Interface segregation

Select two correct answers

1. Which of the following principles aim to reduce coupling in software design?
    a. Single responsibility principle
    b. Dependency inversion principle
    c. Open/closed principle
    d. Law of demeter
2. Which design principles encourage writing code that is easy to extend without modifying existing code?
    a. Open/closed principle
    b. Single responsibility principle
    c. Dependency inversion principle

d. Encapsulation

Answers

This section provides the answers to the given questions.

Find the odd one out

1. Least knowledge
2. Boat anchor

Select two correct answers

1. Dependency inversion principle, law of demeter
2. Open/closed principle, dependency inversion principle

Join our Discord space

Join our Discord workspace for latest updates, offers, tech happenings around the world, new releases, and sessions with the authors:

https://discord.bpbonline.com

Index

A

- abstract factory 50, 51
- abstraction 31
- adapter
  - benefits 106
  - building 99
  - concrete implementations 101
  - drawbacks 106
  - implementation 102-104
  - interfaces 100, 101
  - scenarios 106
  - testing 104-106
- adapter design pattern 98, 99
- AND operation 163
- anti-patterns 270
  - boat anchor 272
  - caching irregularities 271
  - copy and paste 272
  - god object 271
  - hard core 272
  - interface for constants 270, 271
  - no silver bulltes 273
- application programming interface (API) 9, 221

B

- behavioral design patterns 7, 258
  - command design pattern 9
  - observer design pattern 8
  - state design pattern 9
  - strategy design pattern 8
  - use cases 9
- bridge design pattern 146, 147
  - advantages 153
  - concrete transaction modes 150
  - concrete transactions 151
  - drawbacks 153
  - existing implementation 148, 149

  - implementing 148
  - scenarios 153, 154
  - testing 152
  - transaction 149
  - transaction mode 149
- builder design pattern 58
  - advantages 67
  - drawbacks 67
  - scenarios 68
- builders implementation 59-61
  - concrete builders 62, 63
  - deployment manager 64
  - interface 61, 62
  - testing 65-67

C

- central processing unit (CPU) 94
- class patterns 259
- command design pattern 222-224
  - benefits 230
  - client 228, 229
  - command 226, 227
  - drawbacks 230
  - implementing 224
  - invoker 227, 228
  - receiver 224-226
  - scenarios 231
- composite design pattern 171-173
  - benefits 178
  - derived types 174-176
  - drawbacks 178
  - implementing 173, 174
  - product interface 174
  - scenarios 179
  - testing 176, 177
- creational design patterns 3, 4, 257
  - builder design pattern 4
  - factory design pattern 5
  - object pool design pattern 4
  - singleton design pattern 4
  - use cases 5
- Creator class 46

D

- data analysis 157
- decorator
  - base decorator 113, 114

  - building 112
  - concrete decorators 114, 115
  - interface 112
  - subject 113
  - testing 115-118
- decorator design pattern 111, 112
  - advantages 118
  - drawback 118
  - scenarios 118, 119
- dependency injection 49
- dependency inversion principle 263, 264
- deserialization 26
- design patterns 2
  - behavioral design patterns 7, 8
  - classification 3
  - core attributes 2
  - creational design patterns 3, 4
  - structural design patterns 5-7
- design principles 260
  - dependency inversion 263, 264
  - DRY principle 260-262
  - encapsulate what varies 264, 265
  - favor interface over implementation 266, 267
  - interface segregation principle 268, 269
  - least knowledge principle 269, 270
  - open close principle 265, 266
  - single responsibility principle 262, 263
- DRY principle 260-262

E

- encapsulate what varies principle 264, 265
- encapsulation 264
- enum
  - as singleton 25, 26

F

- facade 134
  - advantages 142
  - all for one 136-138
  - building 136
  - drawbacks 142
  - one for all 138-141
  - scenarios 142
- facade design pattern 133-136
- factories
  - building 32-34
- factory design patterns

  advantages 51, 52
  drawbacks 52
  scenarios 52
factory method pattern 45, 46
favor interface over implementation 266, 267
filter design pattern 158
  complex filters 162
  filter interface 159, 160
  implementing 159
  logical operations 162, 163
  simple filters 161, 162
  testing 165-169

H

hooks 184

I

incompatible interfaces 98
individual contributors (ICs) reporting 172
interfaces
  interacting with 32
interface segregation principle 268, 269

J

Java Virtual Machine (VM) 4

L

least knowledge principle 269, 270
logger class 16
  log level, adding 17
  using 17, 18
logical operations 162, 163
  AND 163, 164
OR 164

M

multiple factories 40-42
  abstract response 43, 44
  response in French dialect 42, 43
  second run, of response factory 44

O

object construction
  delegating 56-58

- representational state 59

object patterns 259

object pool 74
- advantages 83
- characteristics 74
- destruction 78, 79
- drawback 84
- implementation 75-77
- implementations, varying 81
- initialization 77, 78
- object release 82, 83
- objects, acquiring 79, 80
- objects, releasing 80, 81
- partial initialization 81, 82
- scenarios 84
- using 75

object pool design pattern 72-74

objects 15

observer design pattern 194-196
- benefits 203
- drawbacks 203
- implementations, varying 200
- implementing 196
- interfaces 196
- pull mechanism 202
- push mechanism 201, 202
- scenarios 203
- subject 197-200
- testing 200

open close principle 265, 266

OR operation 164

P

pipeline design pattern 246, 247
- benefits 254
- concrete tasks 249-251
- drawbacks 254
- implementing 248
- pipeline, constructing 251, 252
- scenarios 255
- task interface 248
- testing 253

placeholders 184

prototype 87, 88
- advantages 94
- concrete implementations 89, 90
- drawbacks 94

- interface 89
- scenarios 94, 95
- testing 90, 91

prototype design pattern 88
- implementing 89

prototype registry 91-94
- importance 92

proxy 122
- advantages 129
- building 125
- drawbacks 129
- interface 125
- scenarios 129
- subject 125-127
- testing 128

proxy class 126

proxy design pattern 122, 123

proxy types 123
- protection proxy 124
- remote proxy 124
- virtual proxy 124

R

reflection
- on singleton objects 24, 25

Reflection API 24

response dialect factory 46-48

response factory 35-38
- first run 38, 39
- second run 44
- third run 49

ResponseFactory class 36

ResponseGenerator class 41

reusability 72

reusable objects 71, 72

S

serialization 26

simple factory 39, 40
- multiple factories 40, 41

single responsibility principle 262, 263

singleton 18

singleton design pattern 18
- advantages 27
- drawbacks 27
- scenarios 28
- structure 18-20

singleton design pattern variations 20
  eager initialization 22, 23
  lazy initialization 20, 21
  multiple instances 21, 22
  thread-safe singleton 23, 24
software design
  good design, benefits 10, 11
  importance 10
state design pattern 208, 209
  benefits 218
  concrete implementations 210-214
  drawbacks 218
  implementing 209
  interface 210
  scenarios 218
  stateful object 214, 215
  testing 216, 217
strategy design pattern 236
  benefits 241
  drawbacks 242
  implementing 236
  interface 237
  scenarios 242
  strategies 237, 238
  strategies. invoking 240
  strategies. selecting 239
  strategies. testing 240
structural design patterns 5, 6, 258
  adapter design pattern 6
  decorator design pattern 6
  facade design pattern 7
  proxy design pattern 7

T

template 182, 183
template method design pattern 183, 184
  benefits 189
  building 185
  drawbacks 189
  implementations, varying 186, 187
  scenarios 189, 190
  testing 187, 188
thread-safe singleton 23, 24
TransactionMode 148

U

Unified Modelling Language (UML) diagram 93

Unified Payments Interface (UPI) 145

W

wrapper 110